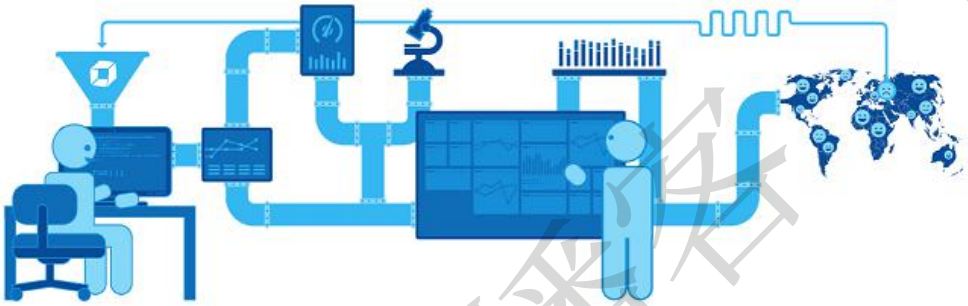
python面试提高篇

部署课程



作者：传智播客-王老师

版本：V 1.0

文档编号：[20181020]

日期：2018年10月30日

目录

第 1 章 部署基础知识

1.1 部署基础

1.1.1 项目生命周期

世间万物皆有其生命，软件项目也是如此。随着互联网的发展，软件项目的生命周期也发生了很大的变化，为了更好的让大家理解软件项目，项目生命周期有狭义[具体]、广义[缘起/缘灭]之分，我们一般所说的项目生命周期主要指的是狭义的项目生命周期，我们以传统的软件项目为例进行介绍。

传统项目生命周期

对于传统软件项目来说，它主要包含以下五个阶段：

1 调研阶段

目的：居安思危

人员：相关人员，侧重于产品经理

节点：多角度思路/方案，最后领导拍板

2 设计阶段

目的：方案可视化

人员：产品团队主导，开发、测试、运维参与

节点：产品需求文档、项目里程表

3 开发阶段

目的：方案运行

人员：开发团队为主，运维团队参与

节点：阶段项目正常运行

4 测试阶段

目的：保证项目功能完善

人员：测试团队为主，运维、开发参与

节点：项目功能符合要求

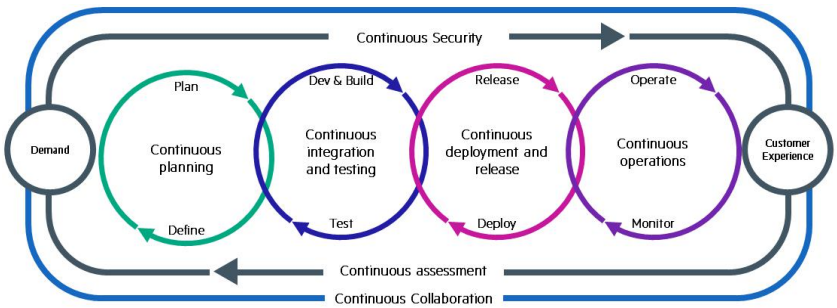
5 运营阶段

目的：项目部署+运行维护

人员：运营团队为主、开发/产品团队参与

节点：项目终止、功能迭代等

新型项目生命周期



1.1.2 项目部署

对于一个软件项目的部署来说，我们主要考虑两方面：

部署流程--基于项目功能，设计合理的部署方案；部署项目--结合部署方案，实现顺畅的项目部署

我们在这一节主要关注部署流程，而部署项目相关知识我们在第4章来进行学习。

关于部署流程，主要包含以下两个方面：部署方案、部署环境

部署方案

为了实现顺畅的项目部署，我们一般会在部署操作的时候，按照既定的部署方案有计划的实施下去。那么部署方案是怎么设计出来的呢？

1 分析项目的产品需求文档，获取项目的主旨，定好部署方案的方向

2 分析项目开发文档，按照功能边界，设计部署的结点
3 分析边界功能，调研功能软件，合理的取舍，选符合当前业务场景的
4 梳理项目部署涉及到的部署软件实现方案，根据2确定的结点，确定初版部署方案
5 根据项目实际情况，调整优化并确定项目部署方案。
附：初版部署方案如何确定最终项目部署方案，请关注第4章内容。

部署环境

根据我们对项目生命周期的理解，一个项目需要经历多个团队的工作，才能最终形成一个成品，也就是说开发好的项目需要在多个阶段环境中按照我们的部署方案部署项目，而每个阶段的环境都是有特殊作用的，那么接下来我们来学习一下项目发布过程中所涉及到的五个环境：

个人开发环境

工作人员：自己
工作平台：个人笔记本、公司配的电脑
平台特点：环境是自己配的，团队中不同的个人开发环境可以不一样
工作内容：项目的子模块，子功能
完成标准：完成领导安排的内容[项目的功能子模块开发]

公司开发环境

工作人员：开发团队
工作平台：公司内部服务器
平台特点：服务器环境和线上的服务器环境完全一致
工作内容：项目子模块间的功能联调
完成标准：项目阶段开发、调试完成

项目测试环境

工作人员：测试团队
工作平台：公司内部服务器
平台特点：服务器环境和线上的服务器环境完全一致
工作内容：项目功能/非功能/探索等测试
完成标准：项目阶段功能正常运行

项目预发布环境

工作人员：运维团队
工作平台：公司线上服务器组中的一台
平台特点：服务器环境和线上的服务器环境完全一致
工作内容：特殊功能测试(比如支付)、数据压力测试、其他安全测试等
完成标准：项目阶段功能正常运行，最后一道防线

项目线上环境

工作人员：运维团队
工作平台：公司线上服务器组
平台特点：标准线上的服务器环境
工作内容：代码部署和维护
完成标准：项目正常运行

第 2 章 Nginx进阶

2.1 Nginx快速入门

2.1.1 Nginx简介

Nginx(发音同 engine x)是一款基于异步框架的轻量级/高性能的Web 服务器/反向代理服务器/缓存服务器/电子邮件(IMAP/POP3)代理服务器，并在一个BSD-like 协议下发行。由俄罗斯的程序设计师Igor Sysoev(伊戈尔·赛索耶夫)所开发，最初供俄国大型网站Rambler.ru及搜寻引擎Rambler使用。

Nginx特点

优点：

高并发量：基于 epoll/kqueue 模型开发，支持高并发量，官方说其支持高达 5w 并发连接数的响应

内存消耗少：善于处理静态文件，相较于其他web(比如：apache)，占用更少的内存及资源

简单稳定：配置简单(一个conf文件)，运行简单(nginx命令)，而且运行稳定

模块化程度高：功能模块插件化设计，可以自由配置相应的功能。

支持Rwrite重写规则：能够根据域名、URL等请求关键点，实现定制化的高质量分发。

低成本：Nginx的负载均衡功能很强大而且免费开源，相较于几十万的硬件负载均衡器成本相当低。

支持多系统：Nginx代码完全用C语言从头写成，可以在各系统上编译并使用。

缺点：

动态处理差：nginx善于处理静态文件，但是处理动态页面相较于Apache之类重量级的web软件能力稍欠缺。

rewrite弱：虽然nginx支持rewrite功能多，但是相较于Apache之类重量级的web软件能力稍欠缺。

2.1.2 Nginx部署

Nginx软件部署

安装Nginx软件

方法一：快

```
apt-get install -y build-essential libssl-dev libtool libpcre3 libpcre3-dev make openssl zlib1g-dev
apt-get install nginx -y
```

方法二：定制标准高

编译安装，请关注后续"shell自动化运维"课程

检查效果：

```
netstat -tnulp | grep nginx
浏览器查看
```

服务相关命令

```
systemctl start|stop|reload|... nginx
/etc/init.d/nginx start|stop|reload|...
/usr/sbin/nginx ...
nginx -V
```

移除相关命令：

查看和nginx相关软件

```
dpkg --get-selections|grep nginx
```

移除nginx，包括相关文件

```
apt-get --purge remove nginx
apt-get --purge remove nginx-common
apt-get --purge remove nginx-core
```

Nginx配置简介

nginx软件目录：

工作目录：/etc/nginx

执行文件：/usr/sbin/nginx

日志目录：/var/log/nginx

启动文件：/etc/init.d/nginx

web目录：/var/www/html/，首页文件是index.nginx-debian.html

/usr/share/nginx/html/ 首页文件是index.html

nginx配置文件：

默认文件：

/etc/nginx/nginx.conf

其他目录：

/etc/nginx/{sites-available/sites-enabled/conf.d}

文件结构：

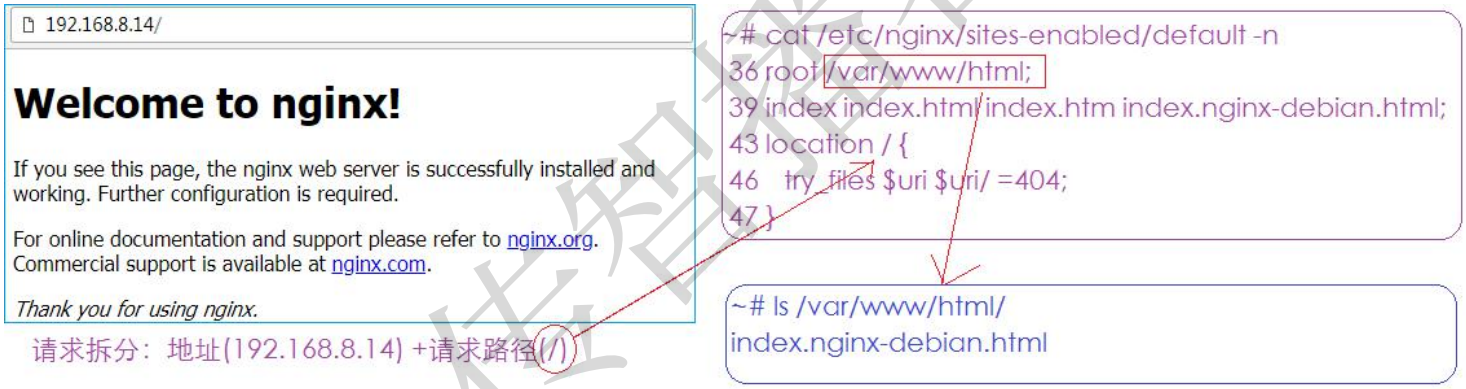
全局配置段

http配置段

server配置段　　项目或者应用

location配置段　　url配置

Nginx访问原理



2.1.3 配置详解

全局配置段

主要是全局性的和服务级别的属性配置，常见的主要有以下几种设置：

user	设置使用用户(worker)
worker_processes	进行增大并发连接数的处理 跟 cpu 保持一致 八核设置八个
error_log	nginx 的错误日志
pid	nginx 服务启动时候 pid
events	定义事件相关的属性
worker_connections	一个进程允许处理的最大连接数
use	定义使用的内核模型

http配置段

主要配置server通用的一些配置

```
include mime.types;                          # 文件扩展名与文件类型映射表
default_type application/octet-stream;       # 默认文件类型
sendfile on;                                 # 开启高效文件传输模式。
autoindex on;                                # 开启目录列表访问，合适下载服务器，默认关闭。
```

```
tcp_nopush on;                          # 防止网络阻塞
tcp_nodelay on;                         # 防止网络阻塞
keepalive_timeout 120;                  # 长连接超时时间，单位是秒
gzip on;                                # 开启 gzip 压缩输出
```

Server常见配置属性

常见样式

```
server {
    listen 端口;
    server_name 主机名;
    ...
}
```

server配置段最重要的属性是listen和server_name。它们都是用于匹配并处理请求的。

listen属性

作用：定义Server监听的ip和port，当ip/port匹配时候才进行下一步匹配

表现形式：

形式	描述	示例	完整示例
IP:Port	地址精确表示样式	listen 10.10.10.10:99	listen 10.10.10.10:99
IP	自动监听 IP:80地址	listen 10.10.10.10	listen 10.10.10.10:80
Port	自动监听 全地址:Port	listen 99或 [::]:99	listen 0.0.0.0:99
default_server	自动使用默认的地址	listen default_server	listen localhost:80

使用原则：

首先将所有样式补全成IP:Port,然后匹配,匹配Server多，那么接着使用Server_name匹配

server_name属性

作用：定义Server监听的域名，当域名匹配时候才进行下一步操作

表现形式：

格式	完整样式	前缀正则样式	后缀正则样式	禁止非法域名或IP
形式	www.example.com	*.example.com	www.example.*	_

使用原则：

优先使用完整样式,然后使用前缀正则样式,最后使用后缀正则样式,如果正则样式相同的时候,匹配最长,否则就走非法规则。

非法域名/IP，表示请求到该主机上一个不存在的IP或者域名

root属性

作用：定义Server相应请求的html文件所在路径

表现形式：

```
root /var/www/html;
```

index属性

作用：定义响应请求后返回的文件名称或格式

表现形式：

```
index index.html index.htm index.nginx-debian.html;
```

return属性

作用：定义响应请求后返回的http状态码

表现形式：

```
return      444;
```

location常见配置属性

location主要是根据Server匹配到的请求路径和关键字去响应和处理。

语法：

```
location optional_modifier location_match { ... }
```

```
location @name { ... }
```

其中：optional_modifier是匹配条件，location_match是匹配的样式，{}是要执行的操作。匹配条件主要有两种：正则/前缀字符。

匹配规则

正则匹配

类型	含义	匹配方式	优先级	样式
~|!~	普通正则-敏感|不敏感	正则符号	3	location ~ .(jpe?g)$ {}
~*|!~*	普通正则-不敏感|敏感	正则符号	3	location ~* .(jpe?g)$ {}

普通匹配

类型	含义	匹配方式	优先级	样式
=/路径	精确匹配	前缀	1	location = /image {}
^~	优先匹配	前缀	2	location ^~ /page {}
@	内部重定向	前缀		location @name {}
空 /	通用匹配	前缀	4	location / {}

使用原则：

前提：根据请求url，获取uri即除了域名/IP之外的部分，用于location匹配

如果有精确匹配，即 =/路径，找到匹配项后，结束匹配。

location = 路径 {} 或者 location 完整路径 {}

如果有优先匹配，即 ^~，找到匹配项后，结束匹配。

location ^~ 路径

如果有正则匹配，即 ~|!~|~*|!~*，找到匹配项后，不会终止继续匹配，直到找到合适的

location ~* 正则字符 {}

如果匹配到多个，则使用location_match最长的。

匹配示例

常见示例：

```
location = / {
    #精确规则A
}
location = /login {
    #精确规则B
}
location ^~ /static/ {
    #优先规则C
}
location ~ \.(gif|jpg|png|js|css)$ {
    #正则规则D
}
location ~* \.png$ {
    #正则规则E
}
location !~ \.xhtml$ {
    #正则规则F
}
location !~* \.xhtml$ {
    #正则规则G
}
location / {
    #通用规则H
}
```

访问效果如下：

访问根目录/， 比如http://a.com/ 将匹配规则A

访问 http://a.com/login 将匹配规则B

访问 http://a.com/static/a.html 将匹配规则C

访问 http://a.com/a.gif, http://a.com/b.png 规则D和E均适合，按顺序优先使用规则D，而 http://a.com/static/c.png 则优先匹配到规则C

访问 http://a.com/a.PNG 则匹配规则E，因为规则E不区分大小写。

访问 http://a.com/a.XHTML 使用规则F。

访问 http://a.com/category/id/1111 则最终匹配到规则H。

@name示例

```
@用来定义一个命名 location。主要用于内部重定向，不能用来处理正常的请求。其用法如下：
location / {
    try_files $uri $uri/ @custom
```

```
    }
    location @custom {
        # ...do something
        # custom 命名的 location 中不能再嵌套其它的命名 location
    }
```

关于URL尾部的/有如下注意事项：

1 location中的location_match字符有无"/"不受影响。/user/等同/user。

2 对于访问网站域名(http://sswang.com/)，尾部有无"/"不受影响。因为浏览器会自动补全"/"。

3 对于访问网站域名后面的路径(http://sswang.com/other/)。尾部的"/"很重要。

URL尾部的"/"表示目录，没有"/"表示文件，而且文件找不到的话，会发生重定向。

/other/：表示服务器会自动去该目录下找对应的默认文件。

/other：表示服务器会先去找other文件，找不到的话会将other当成目录，重定向到/other/，去该目录下找默认文件。

location常见动作：

在location内部常用的功能属性非常多，常见的基本属性、临时跳转、访问控制、目录列表等。

基本属性

```
    location / {
        root   /var/www/html;                    # 指定响应请求的文件所在路径
        index  index.php index.html index.htm;   # 指定响应请求的默认文件名称
        expires 7d;                              # 指定响应请求的文件过期时间，一般用于静态文件
        try_files $uri $uri/ =404;               # 如果 root 指定的路径下有查找的文件，就返回，否则报错
    }
```

临时跳转

```
    location = /test/ {
        return 302 http://sswang.com/;           # 访问旧 url 的时候，临时跳转到新 url，两个 url 均不失效
    }
```

访问控制

```
    location /nginx-status {
        stub_status on;                          # 开启 nginx 的状态页面，默认关闭
        allow 192.168.8.14;                      # 允许的访问地址
        deny all;                                # 默认进制所有访问
    }
    注意：
        该功能依赖于 ngx_http_stub_status_module 模块(默认没有安装，需要定制化安装)
```

目录列表

```
    location /upload {
        alias   /var/www/upload;     # 指定查看文件列表路径(绝对路径)
        autoindex on;                # 开启目录自动索引
        autoindex_exact_size off;    # 默认 on，显示文件确切大小(bytes)。off 表示显示文件的大概大小(kB/MB/...)
        autoindex_localtime on;      # 默认 off，显示的文件时间为 GMT 时间。on 表示显示文件的服务器时间
    }
    注意：
        该 alias 指定的目录下，不允许出现 index 属性指定的文件。
```

root VS alias

root 和 alias 所起的作用都是指定响应请求所用文件的路径，只是他们有些许的区别

root　表示 location 匹配内容的相对路径

alias 表示 一个绝对路径,而且必须以"/"结尾

一般情况下，在location /中配置root，在location /other中配置alias

效果一：

```
location /img/ {
    alias /var/www/image/;
}
```

效果二：

```
location /img/ {
        root /var/www/image;
}
```

效果一：访问http://localhost/img/，nginx找/var/www/image/目录下的文件

效果二：访问http://localhost/img/，nginx找/var/www/image/img/目录下的文件

location核心动作

Nginx的配置语法灵活，可控制度非常高。在0.7以后的版本中加入了一个try_files指令，配合命名location，可以部分替代原本常用的rewrite配置方式，提高解析效率。

指令语法

```
try_files file ... uri
try_files file ... =code
```

作用：

响应时按顺序查找file，找到则返回file内容，否则的话进行内部重定向(uri)或返回状态码(code)。

常见示例

```
如果能找到指定的 uri 那么就返回相应的内容，否则的话返回错误状态码 404
try_files $uri $uri/ =404;
如果能找到指定的文件 1/2.html 那么就返回相应的内容，否则的话返回 6.html 文件内容
try_files 1.html 2.html /6.html;
如果能找到指定的 uri 那么就返回相应的内容，否则的话就内部重定向到后端名称为@backup 的 location
try_files $uri @backup;
```

注意：

如果最后一个地址是一个uri的话，那么这个uri必须是存在的，否则的话就出事了

2.2 Nginx进阶知识

2.2.1 反向代理

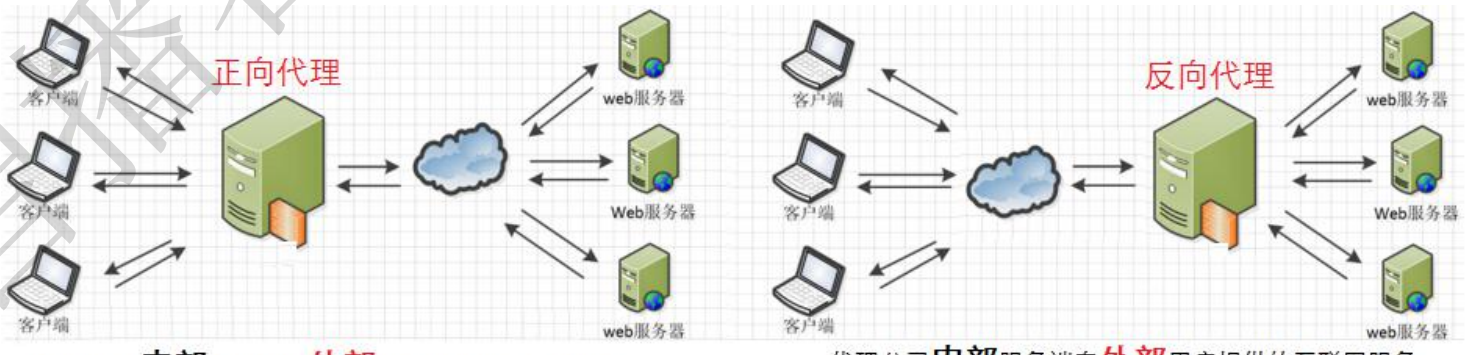
代理是什么？

简单来说，我找一个中间人，代替我去做一件事情，只要他给我结果就可以。

代理一般分为两种：正向代理、反向代理

正向代理&反向代理

示意图



代理公司**内部**客户端向**外部**服务端请求互联网服务　　代理公司**内部**服务端向**外部**用户提供的互联网服务

区别

从用途上来讲：

正向代理-为局域网客户端向外访问Internet服务。可以使用缓冲特性减少网络使用率。

反向代理-为局域网服务器向外提供Internet服务。可以使用负载平衡提高客户访问量。还可以基于高级URL策略和管理技术对服务进行高质量管控。

从安全性来讲：

正向代理-必须采取安全措施确保内网客户端通过它访问外部网站。隐藏客户端的身份

反向代理-对外提供服务是透明的，客户端并不知道自己访问的是一个代理。隐藏服务端的身份

nginx 代理模块

官方介绍

官方资料：http://www.nginx.cn/doc/standard/httpproxy.html

官方的代理属性很多，我们主要介绍proxy_pass和proxy_set_header属性

官方代码示例

```
location / {
: proxy_pass       http://localhost:8000;       # 设定请求跳转后的地址,可以使用 hostname 或 IP:Port 形式
: proxy_set_header  X-Real-IP  $remote_addr;    # 后端请求携带原始请求的真实 IP 地址
}
```

属性详解：

proxy_pass 指令设置被代理服务器的地址和被映射的URI，地址可以使用主机名或IP加端口号的形式

proxy_pass关键点

proxy_pass后面的路径最后的/作用很重要!!!

示例代码：

```
location /html/ {
    1 proxy_pass http://proxy.com;
    2 proxy_pass http://proxy.com/;
}
```

假设我们访问的url是 http://domain.com/html/test.js，如何理解上述两种proxy_pass的区别呢？

```
对于 1 来说 proxy.com 后面没有"/"，表示"/html/" 请求(包括自己)后续的路径及其参数等关键字都由 http://a.com/来处理，代理后的样式如下：
http://proxy.com/html/test.js
对于 2 来所 proxy.com 后面有"/"，表示"/html/" 请求后续的路径及其参数等关键字都由 http://a.com/来处理，代理后的样式如下：
http://proxy.com/test.js
```

nginx代理实践

代理的配置文件

```
~# vim /etc/nginx/conf.d/proxy.conf
server {
 listen 192.168.8.14:80;
 server_name www.sswang.com;
 location / {
  proxy_pass http://192.168.8.14:9999/hello/;
 }
}
```

后端服务配置文件

```
~# vim /etc/nginx/conf.d/hello.conf
server {
  listen 192.168.8.14:9999;
  location /hello/ {
    alias /var/www/html/hello/;
```

```
        try_files $uri $uri/ =404;
      }
    }
```

准备后端服务文件

```
mkdir -p /var/www/html/hello/
echo '<h1>proxy_backend</h1>' > /var/www/html/hello/index.html
```

检查nginx配置后重载服务

```
/usr/sbin/nginx -t
systemctl reload nginx
netstat -tnulp | grep nginx
```

查看效果

2.2.2 负载均衡

负载均衡是什么？

我们之前使用proxy_pass的方式实现了nginx代理请求到后端的效果，随着我们的网站访问量越来越多，一个后端就不现实了，那么接下来我们应该如果在访问量日渐增大的情况下，满足线上业务的稳定呢？

解决方法就是：负载均衡

负载均衡简单说来人多力量大，打群架。

在nginx中的负载均衡主要有两种：四层负载(IP:Port)、七层负载(http://xxx)

nginx upstream模块

官方介绍

官方资料：http://www.nginx.cn/doc/standard/httpupstream.html

官方的代理属性很多，我们主要介绍upstream和ip_hash属性

官方代码示例

```
upstream backend  {
  server backend1.example.com weight=5;
  server backend2.example.com:8080;
  server unix:/tmp/backend3;
}

server {
  location / {
    proxy_pass  http://backend;
  }
}
```

属性详解：

upstream 主要是定义一个后端服务地址的集合列表，每个后端服务使用一个server命令表示

upstream {} 和 Server {} 两部分内容属于平级关系。

后端服务状态

在upstream模块中，可以使用server命令指定后端服务器的地址，同时还可以设置后端服务器在负载均衡调度中的状态，常用的状态有以下几种：

down： 表示当前server主机暂时不参与负载均衡。

backup：后备主机，当所有非backup机器出现故障或者繁忙的时候，才会请求backup机器。

max_fails：允许请求的最大失败数，默认为1，配合fail_timeout一起使用

fail_timeout：经历max_fails次失败，暂停服务的时间，默认为10s。

nginx负载均衡实践

负载均衡配置文件

```
~# vim /etc/nginx/conf.d/upstream.conf
upstream backends {
  server 192.168.8.14:10086;
  server 192.168.8.14:10087;
  server 192.168.8.14:10088;
}
server {
  listen 80;
  server_name localhost;
  location / {
    proxy_pass http://backends;
  }
}
```

后端代理配置文件

```
~# vim /etc/nginx/conf.d/backend.conf
server {
  listen 192.168.8.14:10086;
  location / {
    root /var/www/html/hello/;
    try_files $uri $uri/ =404;
  }
}
server {
  listen 192.168.8.14:10087;
  location / {
    root /var/www/html/nihao/;
    try_files $uri $uri/ =404;
  }
}
server {
  listen 192.168.8.14:10088;
  location / {
    root /var/www/html/huanying/;
    try_files $uri $uri/ =404;
  }
}
```

准备后端服务文件

```
mkdir -p /var/www/html/hello/
echo '<h1>backend_hello</h1>' > /var/www/html/hello/index.html
mkdir -p /var/www/html/nihao/
echo '<h1>backend_nihao</h1>' > /var/www/html/nihao/index.html
mkdir -p /var/www/html/huanying/
echo '<h1>backend_huanying</h1>' > /var/www/html/huanying/index.html
```

检查nginx配置后重载服务

```
/usr/sbin/nginx -t
```

```
systemctl reload nginx
netstat -tnulp | grep nginx
```

查看效果

```
~# for i in {1..100};do curl http://192.168.8.14;done
<h1>backend_hello</h1>
<h1>backend_nihao</h1>
<h1>backend_huanying</h1>
...
<h1>backend_hello</h1>
<h1>backend_nihao</h1>
<h1>backend_huanying</h1>
```

负载均衡调度算法

官方资料：http://nginx.org/en/docs/http/ngx_http_upstream_module.html#example

Nginx提供的负载均衡策略有两种：

内置策略：nginx自带的算法

雨露均沾型：轮训、加权轮训、哈希

定向服务型：ip_hash、least_conn、cookie、route、lean、

商业类型：ntlm、least_time、queue、stick

扩展策略：各种结合业务场景自定义的算法或者第三方算法

自定义算法

第三方算法：**fair、url_hash**

常用算法简介：

```
轮询(默认)：请求按顺序逐一分配到不同的后端服务器。
weight：指定轮询权重，值越大，分配到的几率就越高，适用于后端服务器性能不均衡情况。
ip_hash：按访问 IP 的哈希结果分配请求，分配后访客访问固定后端服务器，有效的解决动态网页会话共享问题。
fair：基于后端服务器的响应时间来分配请求，响应时间短的优先分配。
url_hash：按访问 URL 的哈希结果分配请求，使同 URL 定向到同一台后端服务器，可提高后端缓存服务器的效率。
```

加权轮训实践

修改负载均衡配置文件

```
~# vim /etc/nginx/conf.d/upstream.conf
upstream backends {
  server 192.168.8.14:10086 backup;
  server 192.168.8.14:10087 weight=1;
  server 192.168.8.14:10088 weight=2;
}
...
```

检查nginx配置后重载服务

```
/usr/sbin/nginx -t
systemctl reload nginx
netstat -tnulp | grep nginx
```

查看效果

```
~# for i in {1..100};do curl http://192.168.8.14;done
<h1>backend_nihao</h1>
<h1>backend_huanying</h1>
<h1>backend_huanying</h1>
...
<h1>backend_nihao</h1>
```

```
<h1>backend_huanying</h1>
<h1>backend_huanying</h1>
```

ip_hash实践

修改负载均衡配置文件

```
~# vim /etc/nginx/conf.d/upstream.conf
upstream backends {
  ip_hash;
  server 192.168.8.14:10086 ;
  ...
}
...
```

检查nginx配置后重载服务

```
/usr/sbin/nginx -t
systemctl reload nginx
netstat -tnulp | grep nginx
```

查看效果

```
~# for i in {1..100};do curl http://192.168.8.14;done
<h1>backend_huanying</h1>
...
<h1>backend_huanying</h1>
```

2.2.3 日志解析

日志功能简介

日志简介

Nginx默认提供了两个日志文件 access.log和error.log，通过access.log可以得到用户请求的相关信息；通过error.log可以获取某个web服务故障或其性能瓶颈等信息。

而且nginx的日志支持定制化格式，这样我们就可以根据实际的业务情况更好的高效工作。最常见的场景就是获取客户端的IP，记录用户访问量。

官方介绍：http://nginx.org/en/docs/http/ngx_http_log_module.html

基本配置

```
# cat /etc/nginx/nginx.conf -n
    40      access_log /var/log/nginx/access.log;
    41      error_log /var/log/nginx/error.log;
注意：
    nginx 日志属性设置的完整格式是：
        属性名称 access_log
        存储位置 /var/log/nginx/access.log
        日志格式 位置为空表示使用默认的 combined 日志格式。它是通过 log_format 设置的
```

默认日志格式

```
log_format combined '$remote_addr - $remote_user [$time_local] '
                    '"$request" $status $body_bytes_sent '
                    '"$http_referer" "$http_user_agent"';
```

注意：

log_format是有一批nginx内置变量组合而成的。

日志样式：

```
# tail /var/log/nginx/access.log
```

```
    192.168.8.14 - - [12/Nov/2018:08:24:18 -0800] "GET /favicon.ico HTTP/1.0" 404 580 "-" "Mozilla/5.
0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2883.87 Safari/537.36"
```

nginx常用内置变量

nginx常用的内置变量主要是用来分析日志中的http记录的，我们可以根据内置的变量精确的获取相关的信息

默认变量

$remote_addr	前一台主机的 ip 地址，不一定是真实的客户端 IP
$remote_user	用于记录远程客户端的用户名称（一般为“-”）
$time_local	用于记录访问时间和时区
$request	用于记录请求的 url 以及请求方法
$status	响应状态码，例如：200 成功、404 页面找不到等。
$body_bytes_sent	给客户端发送的文件主体内容字节数
$http_referer	可以记录用户是从哪个链接访问过来的
$http_user_agent	用户所使用的代理（一般为浏览器）

其他常用变量

$request_uri	包含请求参数的原始 URI，不包含主机名
$uri	不带请求参数的当前 URI，不包含主机名
$http_x_forwarded_for	可以记录客户端 IP，通过代理服务器来记录客户端的 ip 地址
$http_x_real_ip	可以记录客户端 IP，通过代理服务器来记录客户端的 ip 地址
$args	这个变量等于请求行中的参数，同$query_string
$host	请求主机头字段，否则为服务器名称。
$scheme	HTTP 方法（如 http，https）
$document_uri	与$uri 相同
$document_root	当前请求文件配置文件中 html 的根目录即 root 值
$request_filename	当前请求的文件路径，由 root 或 alias 指令与 URI 请求生成

示例：

例：http://localhost:10086/sswang1/sswang2/test.txt	
$host	localhost
$server_port	10086
$request_uri	/sswang1/sswang2/test.txt
$document_uri	/sswang1/sswang2/test.txt
$document_root	/var/www/html
$request_filename	/var/www/html/sswang1/sswang2/test.txt

自定义日志实践

需求：

基于代理方式访问app1应用，日志存放在/var/logs/nginx/app1/access.log，要求能从日志中获取客户端的IP地址

因为是获取代理前面客户端的真实IP，需要nginx开启 --with-http_realip_module 功能，使用nginx -V来检查，ubuntu默认安装的已经开启了该功能。

设置日志格式

```
~# vim /etc/nginx/nginx.conf
   ##
   # Logging Settings
   ##
      log_format proxy_format '$remote_addr - $remote_user [$time_local] '
                '"$request" $status $body_bytes_sent "$http_referer"'
```

```
                  '"$http_user_agent" "$http_x_real_ip" "$http_x_forwarded_for"';
```

负载均衡配置文件

```
~# vim /etc/nginx/conf.d/upstream.conf
upstream backends {
  server 192.168.8.14:10086;
}
server {
  listen 80;
  server_name localhost;
  location / {
    proxy_pass http://backends;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
}
```

后端代理配置文件

```
~# vim /etc/nginx/conf.d/backend.conf
server {
  listen 192.168.8.14:10086;
  root /var/www/html/app1/;
  access_log /var/log/nginx/app1/access.log proxy_format;
  real_ip_header X-Forwarded-For;
  set_real_ip_from 192.168.0.0/16;
  real_ip_recursive on;
  location / {
    try_files $uri $uri/ =404;
  }
}
```

准备后端服务文件

```
mkdir -p /var/www/html/app1/
echo '<h1>backend_app1</h1>' > /var/www/html/app1/index.html
mkdir /var/log/nginx/app1 -p
```

检查nginx配置后重载服务

```
/usr/sbin/nginx -t
systemctl reload nginx
netstat -tnulp | grep nginx
```

查看效果

```
在多台主机上执行如下命令
curl http://192.168.8.14
```

查看日志：

app1日志

```
    192.168.8.1 - - [12/Nov/2018:18:28:46 -0800] "GET / HTTP/1.0" 200 22 "-" "Mozilla/5.0 (Windows N
T 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.99 Safari/537.36" "192.1
68.8.1" "192.168.8.1"
    192.168.8.14 - - [12/Nov/2018:18:29:30 -0800] "GET / HTTP/1.0" 200 22 "-" "curl/7.47.0" "192.168.
8.14" "192.168.8.14"
    192.168.8.15 - - [12/Nov/2018:18:31:43 -0800] "GET / HTTP/1.0" 200 22 "-" "curl/7.29.0" "192.168.
8.15" "192.168.8.15"
```

注意：

因为我们的虚拟机使用的是nat网络模型，所以我们用外部的宿主机来访问的话，是通过VMnat8网卡IP来访问nginx代理的，所以记录的是192.168.8.1

注意：

如果生产中出现了多级代理，

在第一层代理上添加 proxy_set_header X-Real-IP $remote_addr;属性

在所有代理上必须添加 proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;属性

真实主机上使用 real_ip_header X-Forwarded-For;属性

第 3 章 Docker快速入门

3.1 docker快速入门

学习目标：

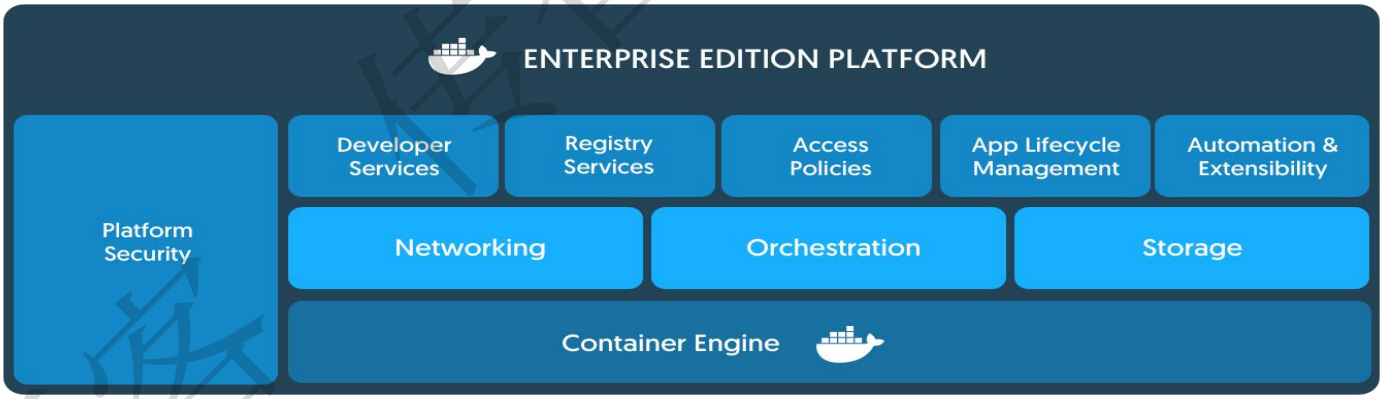
了解 docker特点和场景

3.1.1 docker是什么

这一节，我们从定义、场景、历史这三个方面来学习

docker是什么？

Docker is the company driving(推动) the container movement and the only container platform provider to address every application across the hybrid cloud(混合云). Today's businesses are under pressure to digitally transform(数字化转型) but are constrained(限制) by existing applications and infrastructure while rationalizing an increasingly diverse portfolio of clouds, datacenters and application architectures. Docker enables true independence between applications and infrastructure and developers and IT ops to unlock their potential and creates a model for better collaboration and innovation.



Docker是一个开源的容器引擎，它基于LXC容器技术，使用Go语言开发。

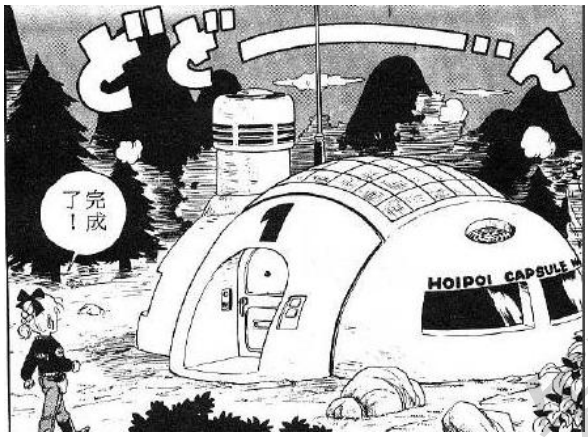
源代码托管在Github上，并遵从Apache2.0协议。

Docker采用C/S架构，其可以轻松的为任何应用创建一个轻量级的、可移植的、自给自足的容器。

简单来说：Docker就是一种快速解决生产问题的一种技术手段。

Docker生活场景：

图一：动画片《七龙珠》里面的胶囊



图二：1号胶囊启动后的效果

官方资料：

Docker 官网：http://www.docker.com

Github Docker 源码：https://github.com/docker/docker

Docker理念

构建：

龙珠里的胶囊，将你需要的场景构建好，装在一个小胶囊里

运输：

随身携带着房子、车子等，非常方便

运行：

只需要你轻轻按一下胶囊，找个合适的地方一放，就ok了

优缺点

优点：

多：　适用场景多

快：　环境部署快、更新快

好：　好多人在用，东西好

省：　省钱省力省人工(123原则)

缺点：

太腻歪人：　依赖操作系统

不善于沟通：依赖网络

不善理财：　银行U盾等场景不能用

3.1.2 部署docker

这一节，我们从软件源配置、基础软件安装、docker安装六个方面来学习

软件源配置

官网参考：

https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/#upgrade-docker-after-using-the-convenience-script

基础软件源

```
cd /etc/apt/
mv sources.list sources.list.bak
vim sources.list
# sohu shangdong
deb http://mirrors.sohu.com/ubuntu/ trusty main restricted universe multiverse
```

```
    deb http://mirrors.sohu.com/ubuntu/ trusty-security main restricted universe multiverse
    deb http://mirrors.sohu.com/ubuntu/ trusty-updates main restricted universe multiverse
    deb http://mirrors.sohu.com/ubuntu/ trusty-proposed main restricted universe multiverse
    deb http://mirrors.sohu.com/ubuntu/ trusty-backports main restricted universe multiverse

    # 163 guangdong
    deb http://mirrors.163.com/ubuntu/ trusty main restricted universe multiverse
    deb http://mirrors.163.com/ubuntu/ trusty-security main restricted universe multiverse
    deb http://mirrors.163.com/ubuntu/ trusty-updates main restricted universe multiverse
    deb http://mirrors.163.com/ubuntu/ trusty-proposed main restricted universe multiverse
    deb http://mirrors.163.com/ubuntu/ trusty-backports main restricted universe multiverse

    # tsinghua.edu
    deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ xenial main restricted universe multiverse
    deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ xenial-updates main restricted universe multiverse
    deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ xenial-backports main restricted universe multiverse
    deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ xenial-security main restricted universe multiverse

    # aliyun
    deb http://mirrors.aliyun.com/ubuntu/ xenial main restricted universe multiverse partner
    deb http://mirrors.aliyun.com/ubuntu/ xenial-updates main restricted universe multiverse
    deb http://mirrors.aliyun.com/ubuntu/ xenial-backports main restricted universe multiverse
    deb http://mirrors.aliyun.com/ubuntu/ xenial-security main restricted universe multiverse

    # neu.edu
    deb http://mirror.neu.edu.cn/ubuntu/ xenial main restricted universe multiverse partner
    deb http://mirror.neu.edu.cn/ubuntu/ xenial-updates main restricted universe multiverse
    deb http://mirror.neu.edu.cn/ubuntu/ xenial-backports main restricted universe multiverse
    deb http://mirror.neu.edu.cn/ubuntu/ xenial-security main restricted universe multiverse
```

安装依赖软件

```
    apt-get update
    apt-get install apt-transport-https ca-certificates curl software-properties-common -y
```

使用官方推荐源

```
    curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
    add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs)
 stable"
```

使用阿里云的源**{推荐1}**

```
    curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key add -
    add-apt-repository "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb_re
lease -cs) stable"
```

使用清华的源**{推荐2}**

```
    curl -fsSL https://mirrors.tuna.tsinghua.edu.cn/docker-ce/linux/ubuntu/gpg | sudo apt-key add -
    add-apt-repository "deb [arch=amd64] https://mirrors.tuna.tsinghua.edu.cn/docker-ce/linux/ubun
tu $(lsb_release -cs) stable"
```

检查

```
apt-get update
```

docker软件安装

查看支持的docker版本

```
apt-cache madison docker-ce
```

安装docker

```
apt-get install docker-ce -y
```

注：

可以指定版本安装docker：

```
apt-get install docker-ce=<VERSION> -y
```

启动docker

```
systemctl start docker
systemctl status docker
注意：
    ubuntu 安装完毕后，默认就开启服务了
```

测试docker

```
docker version
```

网卡区别：

安装前：只有ens33和lo网卡

安装后：docker启动后，多出来了docker0网卡，网卡地址172.17.0.1

docker服务命令格式：

systemctl [参数] docker

参数详解：

start	开启服务
stop	关闭
restart	重启
status	状态

删除docker命令：

```
yum remove docker-ce
rm -rf /var/lib/docker/
rm -rf /etc/docker
```

基本目录

docker基本目录简介

/etc/docker/	docker的认证目录
/var/lib/docker/	docker的应用目录

3.1.3 docker加速器

这一节，我们从加速器简介，加速器配置这两个方面来学习

加速器简介

在国内使用docker的官方镜像源，会因为网络的原因，造成无法下载，或者一直处于超时。所以我们使用 daocloud 的方法进行加速配置。

方法：

访问 daocloud.io 网站，登录 daocloud 账户

点击右上角的 加速器

在新窗口处会显示一条命令，

我们执行这条命令

```
curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://74f21445.m.daocloud.io
```

加速器配置

修改daemon.json文件，增加绿色背景字体内容

```
# cat /etc/docker/daemon.json
{"registry-mirrors": ["http://74f21445.m.daocloud.io"], "insecure-registries": []}
```

注意：

docker cloud加速器的默认内容是少了一条配置，所以我们要编辑文件在后面加上绿色背景的内容，然后再重启docker

重启docker

```
systemctl restart docker
```

3.2 镜像管理

学习目标：

说出 Docker镜像的定义和作用

应用 Docker镜像的基本操作

3.2.1 镜像简介

Docker镜像是什么？

它是一个只读的文件，就类似于我们安装操作系统时候所需要的那个iso光盘镜像，通过运行这个镜像来完成各种应用的部署。

这里的镜像就是一个能被docker运行起来的一个程序。

3.2.2 搜索、查看、获取、历史

这一节，我们从搜索、查看、获取三个方面来学习

搜索镜像

命令格式：

docker search [image_name]

命令演示：

```
docker search ubuntu
```

获取镜像

命令格式：

docker pull [image_name]

命令演示：

```
docker pull ubuntu
docker pull nginx
```

注释：

获取的镜像在哪里？

/var/lib/docker 目录下，具体详见docker仓库知识

查看镜像

命令格式：

docker images <image_name>

命令演示：

```
docker images
```

镜像的ID唯一标识了镜像，如果ID相同，说明是同一镜像。TAG信息来区分不同发行版本，如果不指定具体标记，默认使用latest标记信息

docker images -a 列出所有的本地的images(包括已删除的镜像记录)

查看镜像历史

查看镜像历史命令格式：

```
docker history [image_name]
```

我们获取到一个镜像，想知道他默认启动了哪些命令或者都封装了哪些系统层，那么我们可以使用docker history这条命令来获取我们想要的信息

3.2.3 重命名、删除

这一节，我们从重命名、删除这两个方面来学习。

镜像重命名

命令格式：

```
docker tag [old_image]:[old_version] [new_image]:[new_version]
```

命令演示：

```
docker tag nginx:latest sswang-nginx:v1.0
```

删除镜像

命令格式：

```
docker rmi [image_id/image_name:image_version]
```

命令演示：

```
docker rmi 3fa822599e10
```

注意：

如果一个image_id存在多个名称，那么应该使用name:tag的格式删除镜像

清除状态为dangling的镜像

```
docker image prune
```

移除所有未被使用的镜像

```
docker image prune -a
```

删除部分镜像

```
docker image prune -a --filter "until=24h"
```

3.2.4 导出、导入

这一节，我们从镜像导入、导出两个方面来学习。

导出镜像

将已经下载好的镜像，导出到本地，以备后用。

命令格式：

```
docker save -o [包文件] [镜像]
docker save [镜像1] ... [镜像n] > [包文件]
```

注意：

docker save 会保存镜像的所有历史记录和元数据信息

导出镜像

```
docker save -o nginx.tar sswang-nginx
```

导入镜像

为了更好的演示效果，我们先将nginx的镜像删除掉

```
docker rmi nginx:v1.0
docker rmi nginx
```

导入镜像命令格式：

```
docker load < [image.tar_name]
docker load --input [image.tar_name]
```

注意：

docker load 不能指定镜像的名称

导入镜像文件

```
docker load < nginx.tar
```

3.3 容器管理

学习目标：

说出 Docker容器的定义和作用

应用 Docker容器的基本操作

3.3.1 容器简介

容器是什么？

容器就类似于我们运行起来的一个操作系统，而且这个操作系统启动了某些服务。

这里的容器指的是运行起来的一个Docker镜像。

3.3.2 查看、启动

这一节，我们从容器查看、启动两个方面来学习。

查看容器

命令格式：docker ps

```
# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
          NAMES
```

注意：

管理docker容器可以通过名称，也可以通过ID

ps是显示正在运行的容器， -a是显示所有运行过的容器，包括已经不运行的容器

启动容器

守护进程方式启动容器

命令格式：docker run <参数，可选> [docker_image] [执行的命令]

让Docker容器在后台以守护形式运行。此时可以通过添加-d参数来实现

```
 docker run -d nginx
```

启动已终止的容器

在生产过程中，常常会出现运行和不运行的容器，我们使用 start 命令开起一个已关闭的容器

命令格式：docker start [container_id]

3.3.3 关闭、删除

这一节，我们从容器关闭、删除这两个方面来学习

关闭容器

在生产中，我们会以为临时情况，要关闭某些容器，我们使用 stop 命令来关闭某个容器

命令格式：docker stop [container_id]

关闭容器id

```
docker stop 8005c40a1d16
```

删除容器

删除容器有两种方法：

正常删除 -- 删除已关闭的

强制删除 -- 删除正在运行的

正常删除容器

命令格式： docker rm [container_id]
docker container prune

删除已关闭的容器

```
docker rm 1a5f6a0c9443
```

强制删除运行容器

命令格式：docker rm -f [container_id]

删除正在运行的容器

```
docker rm -f 8005c40a1d16
```

删除部分容器

```
docker container prune --filter "until=24h"
```

拓展批量关闭容器

命令格式：

```
docker rm -f $(docker ps -a -q)
```

3.3.4 进入、退出

这一节，我们从容器进入(三方法)、退出两个方面来学习。

进入容器我们学习两种方法：

1、创建容器的同时进入容器

2、手工方式进入容器

创建并进入容器

命令格式：docker run --name [container_name] -it [docker_image] /bin/bash

```
~]# docker run -it --name sswang-nginx nginx /bin/bash
root@7c5a24a68f96:/# echo "hello world"
hello world
root@7c5a24a68f96:/# exit
exit
```

docker 容器启动命令参数详解：

--name:给容器定义一个名称

-i:则让容器的标准输入保持打开。

-t:让docker分配一个伪终端,并绑定到容器的标准输入上

/bin/bash:执行一个命令

退出容器：

方法一：exit

方法二：Ctrl + D

手工方式进入容器

命令格式：

docker exec -it 容器id /bin/bash

效果演示：

```
docker exec -it d74fff341687 /bin/bash
```

3.3.5 基于容器创建镜像

提交方式：

命令格式：

```
docker commit -m '改动信息' -a "作者信息" [container_id] [new_image:tag]
```

命令演示：

进入一个容器，创建文件后并退出

```
./docker_in.sh d74fff341687
mkdir /sswang
exit
```

创建一个镜像

```
docker commit -m 'mkdir /sswang' -a "sswang" d74fff341687 sswang-nginx:v0.2
```

查看镜像

```
docker images
```

启动一个容器

```
docker run -itd sswang-nginx:v0.2 /bin/bash
```

进入容器进行查看

```
./docker_in.sh ae63ab299a84
ls
```

3.3.6 日志、信息

这一节，我们从日志、详细信息两方面来学习。

查看容器运行日志

命令格式：

```
docker logs [容器id]
```

命令效果：

```
docker logs 7c5a24a68f96
```

查看容器详细信息

命令格式：

```
docker inspect [容器id]
```

命令效果：

查看容器全部信息

```
docker inspect 930f29ccdf8a
```

查看容器网络信息

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' 930f29ccdf8a
```

3.4 仓库管理

学习目标：

说出 Docker仓库的定义和作用

应用 Docker仓库的基本操作

3.4.1 仓库简介

这一节，我们从仓库定义、docker仓库、相关命令这三个方面来学习。

仓库定义

仓库是什么？

仓库就类似于我们在网上搜索操作系统光盘的一个镜像站。

这里的仓库指的是Docker镜像存储的地方。

Docker仓库

Docker的仓库有三大类：

公有仓库：Docker hub、Docker cloud、等

私有仓库：registry、harbor等

本地仓库：在当前主机存储镜像的地方。

相关命令

和仓库相关的命令：

```
docker login [仓库名称]
docker pull [镜像名称]
docker push [镜像名称]
docker search [镜像名称]
```

我们接下来就用registry来部署一个私有的仓库

3.4.2 私有仓库部署

这一节，我们从流程、方案两个方面来学习

创建仓库流程

1、根据registry镜像创建容器

2、配置仓库权限

3、提交镜像到私有仓库

4、测试

实施方案

下载registry镜像

```
docker pull registry
```

启动仓库容器

```
docker run -d -p 5000:5000 registry
```

检查容器效果

```
curl 127.0.0.1:5000/v2/_catalog
```

配置容器权限

```
vim /etc/docker/daemon.json
{"registry-mirrors": ["http://74f21445.m.daocloud.io"], "insecure-registries": ["192.168.8.14:5000"]}
```

注意：

私有仓库的ip地址是宿主机的ip，而且ip两侧有双引号

重启docker服务

```
systemctl restart docker
systemctl status docker
```

效果查看

启动容器

```
docker start 315b5422c699
```

标记镜像

```
docker tag ubuntu-mini 192.168.8.14:5000/ubuntu-14.04-mini
```

提交镜像

```
docker push 192.168.8.14:5000/ubuntu-14.04-mini
```

下载镜像

```
docker pull 192.168.8.14:5000/ubuntu-14.04-mini
```

3.5 数据管理

学习目标：

说出 数据卷、数据容器是什么

应用 数据卷、数据容器的常见操作

docker的镜像是只读的，虽然依据镜像创建的容器可以进行操作，但是我们不能将数据保存到容器中，因为容器会随时关闭和开启，那么如何将数据保存下来呢？

答案就是：数据卷和数据卷容器

3.5.1 数据卷简介

这一节，我们从定义、命令详解 这两个方面来学习。

什么是数据卷？

就是将宿主机的某个目录，映射到容器中，作为数据存储的目录，我们就可以在宿主机对数据进行存储

缺点是：太单一了

docker 数据卷命令详解

```
# docker run --help
。。。
  -v, --volume list            Bind mount a volume (default [])
                               挂载一个数据卷，默认为空
```

我们可以使用命令 docker run 用来创建容器，可以在使用docker run 命令时添加 -v 参数，就可以创建并挂载一个到多个数据卷到当前运行的容器中。

-v 参数的作用是将宿主机的一个目录(绝对路径)作为容器的数据卷挂载到docker容器中，使宿主机和容器之间可以共享一个目录，如果本地路径不存在，Docker也会自动创建。

-v 宿主机文件:容器文件

3.5.2 数据卷实践

这一节，我们从目录实践、文件实践两个方面来学习。

关于数据卷的管理我们从两个方面来说：

1、目录

2、普通文件

数据卷实践 之 目录

命令格式：

```
docker run -itd --name [容器名字] -v [宿主机目录]:[容器目录] [镜像名称] [命令(可选)]
```

命令演示：

创建测试文件

```
echo "file1" > /tmp/file1.txt
```

启动一个容器，挂载数据卷

```
docker run -itd --name test1 -v /tmp:/test1 nginx
```

测试效果

```
~# docker exec -it a53c61c77 /bin/bash
root@a53c61c77bde:/# cat /test1/file1.txt
file1
```

数据卷实践 之 文件

命令格式：

```
docker run -itd --name [容器名字] -v [宿主机文件]:[容器文件] [镜像名称] [命令(可选)]
```

注意：容器里面的文件虽然可以改名，但类型必须和宿主机文件一致

命令演示：

创建测试文件

```
echo "file1" > /tmp/file1.txt
```

启动一个容器，挂载数据卷

```
docker run -itd --name test2 -v /tmp/file1.txt:/nihao/nihao.sh nginx
```

测试效果

```
~# docker exec -it 84c37743 /bin/bash
root@84c37743d339:/# cat /nihao/nihao.sh
file1
```

数据卷实践 之 删除

```
docker volume rm
docker volume prune
```

3.5.3 数据卷容器简介

这一节，我们从定义、命令详解、操作流程这三个方面来学习。

什么是数据卷容器？

将宿主机的某个目录，使用容器的方式来表示，然后其他的应用容器将数据保存在这个容器中，达到大批量应用数据同时存储的目的

docker 数据卷命令详解

```
# docker run --help
。。。
  -v, --volumes-from value      Mount volumes from the specified container(s) (default [])
                                从指定的容器挂载卷，默认为空
```

数据卷容器操作流程

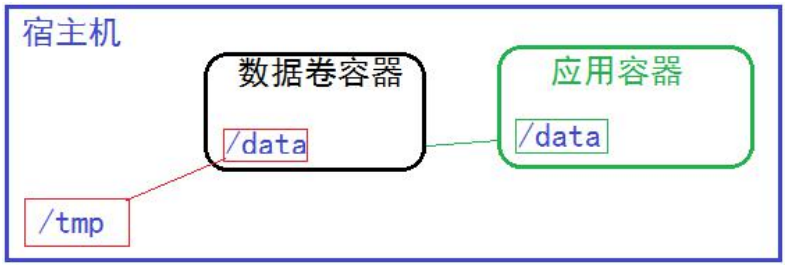
如果使用数据卷容器，在多个容器间共享数据，并永久保存这些数据，需要有一个规范的流程才能做得到：

1、创建数据卷容器

2、其他容器挂载数据卷容器

注意：

数据卷容器不启动



3.5.4 数据卷容器实践

这一节，我们从创建、使用、效果查看三个方面来学习。

数据卷容器实践包括两部分：创建数据卷容器和使用数据卷容器

创建一个数据卷容器

命令格式：

```
docker create -v [容器数据卷目录] --name [容器名字] [镜像名称] [命令(可选)]
```

执行效果

```
docker create -v /data --name v-test nginx
```

创建两个容器，同时挂载数据卷容器

命令格式：

```
docker run --volumes-from [数据卷容器id/name] -tid --name [容器名字] [镜像名称] [命令(可选)]
```

执行效果：

创建 vc-test1 容器

```
docker run --volumes-from 4693558c49e8 -tid --name vc-test1 nginx /bin/bash
```

创建 vc-test2 容器

```
docker run --volumes-from 4693558c49e8 -tid --name vc-test2 nginx /bin/bash
```

确认卷容器共享

进入vc-test1，操作数据卷容器

```
~# docker exec -it vc-test1 /bin/bash
root@c408f4f14786:/# ls /data/
root@c408f4f14786:/# echo 'v-test1' > /data/v-test1.txt
root@c408f4f14786:/# exit
```

进入vc-test2，确认数据卷

```
~# docker exec -it vc-test2 /bin/bash
root@7448eee82ab0:/# ls /data/
v-test1.txt
root@7448eee82ab0:/# echo 'v-test2' > /data/v-test2.txt
root@7448eee82ab0:/# exit
```

回到vc-test1进行验证

```
~# docker exec -it vc-test1 /bin/bash
root@c408f4f14786:/# ls /data/
v-test1.txt  v-test2.txt
root@c408f4f14786:/# cat /data/v-test2.txt
v-test2
```

回到宿主机查看/data/目录

```
~# ls /data/
~#
```

结果证明：

容器间可以共享数据卷你容器，不过数据是保存在数据卷内，并没有保存到宿主机的文件目录中

3.6 网络管理

学习目标：

了解 Docker网络模型及其特点

应用 端口映射常见操作

应用 Docker 常见网络模型

Docker 网络很重要，重要的，我们在上面学到的所有东西都依赖于网络才能工作。我们从两个方面来学习网络：端口映射和网络模式

为什么先学端口映射呢？

在一台主机上学习网络，学习端口映射最简单，避免过多干扰。

3.6.1 端口映射详解

这一节，我们从简介、种类两个方面来学习。

端口映射简介

默认情况下，容器和宿主机之间网络是隔离的，我们可以通过端口映射的方式，将容器中的端口，映射到宿主机的某个端口上。这样我们就可以通过 宿主机的ip+port的方式来访问容器里的内容

端口映射种类

1、随机映射　　-P(大写)

2、指定映射　　-p 宿主机端口:容器端口

注意：

生产场景一般不使用随机映射，但是随机映射的好处就是由docker分配，端口不会冲突，

不管哪种映射都会影响性能，因为涉及到映射

3.6.2 随机映射实践

这一节，我们从随机映射、指定随机映射这两个方面来学习。

随机映射我们从两个方面来学习：

默认随机映射

指定主机随机映射

默认随机映射

命令格式：

```
docker run -d -P [镜像名称]
```

命令效果：

启动一个 nginx 镜像

```
docker run -d -P nginx
```

查看效果

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                    NAMES
930f29ccdf8a        nginx               "nginx -g 'daemon of…"   3 seconds ago       Up 3 seconds        0.0.0.0
:32768->80/tcp      peaceful_pike
```

注意：

宿主机的32768被映射到容器的80端口

-P 自动绑定所有对外提供服务的容器端口，映射的端口将会从没有使用的端口池中自动随机选择，但是如果连续启动多个容器的话，则下一个容器的端口默认是当前容器占用端口号+1

在浏览器中访问 http://192.168.8.14:32768/，效果显示：

注意：

浏览器输入的格式是：　docker容器宿主机的ip:容器映射的端口

指定主机随机映射

命令格式

```
docker run -d -p [宿主机ip]::[容器端口] --name [容器名称] [镜像名称]
```

命令效果

```
docker run -d -p 192.168.8.14::80 --name nginx-2 nginx
```

检查效果

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
       PORTS                     NAMES
5a26e7e8d8a8        nginx               "nginx -g 'daemon of…"   About a minute ago   Up About a mi
nute   192.168.8.14:32769->80/tcp   nginx-2
```

3.6.3 指定映射实践

这一节，我们从指定端口、指定多端口两个方面来学习。

指定端口映射

命令格式：

```
docker run -d -p [宿主机ip]:[宿主机端口]:[容器端口] --name [容器名字] [镜像名称]
```

注意：

如果不指定宿主机ip的话，默认使用 0.0.0.0，

容器端口必须清楚，而且必须写出来

命令实践：

现状我们在启动容器的时候，给容器指定一个访问的端口 1199

```
docker run -d -p 192.168.8.14:1199:80 --name nginx-1 nginx
```

查看新容器ip

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' 0ad3acfbfb76
```

查看容器端口映射

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                    NAMES
0ad3acfbfb76        nginx               "nginx -g 'daemon of…"   37 seconds ago      Up 36 seconds       192.168
.8.14:1199->80/tcp   nginx-1
930f29ccdf8a        nginx               "nginx -g 'daemon of…"   25 minutes ago      Up 25 minutes       0.0.0.0
:32768->80/tcp       peaceful pike
```

查看宿主机开启端口

```
root@admina-virtual-machine:~# netstat -tnulp | grep docker-proxy
tcp        0      0 192.168.8.14:1199       0.0.0.0:*               LISTEN      58951/docker-proxy
tcp6       0      0 :::32768                :::*                    LISTEN      58487/docker-proxy
```

查看浏览器效果：

多端口映射方法

命令格式

```
docker run -d -p [宿主机端口1]:[容器端口1]  -p [宿主机端口2]:[容器端口2] --name [容器名称] [镜像名称]
```

开起多端口映射实践

```
docker run -d -p 520:443 -p 6666:80 --name nginx-3 nginx
```

查看容器进程

```
2b7c9da644c2        nginx               "nginx -g 'daemon of…"   10 seconds ago      Up 10 seconds
      0.0.0.0:6666->80/tcp, 0.0.0.0:520->443/tcp   nginx-3
```

3.6.4 网络管理基础

这一节，我们主要从网络命令、网络模型两方面来学习。

docker网络命令

查看网络命令帮助

```
~# docker network help
...
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks
```

docker的网络模式

bridge模式：

简单来说：就是穿马甲，打着宿主机的旗号，做自己的事情。

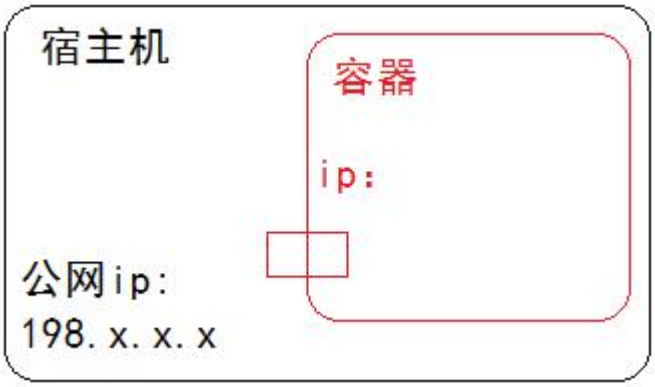
Docker的默认模式，它会在docker容器启动时候，自动配置好自己的网络信息，同一宿主机的所有容器都在一个网络下，彼此间可以通信。类似于我们vmware虚拟机的nat模式。

利用宿主机的网卡进行通信，因为涉及到网络转换，所以会造成资源消耗，网络效率会低。

host模式：

简单来说，就是鸠占鹊巢，用着宿主机的东西，干自己的事情。容器使用宿主机的ip地址进行通信。

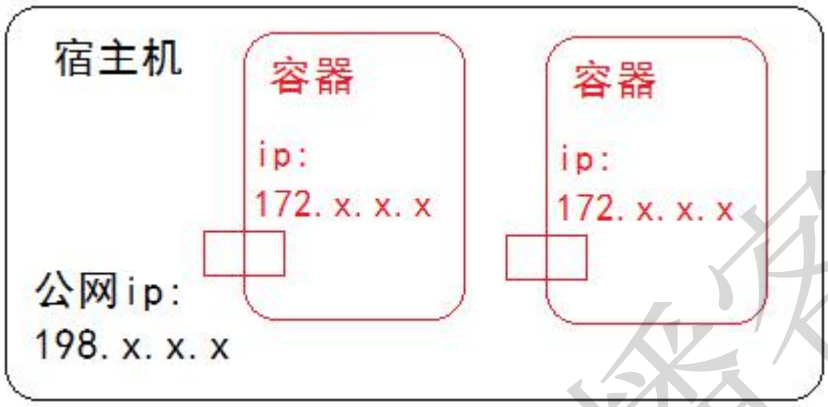
特点：容器和宿主机共享网络



container模式：

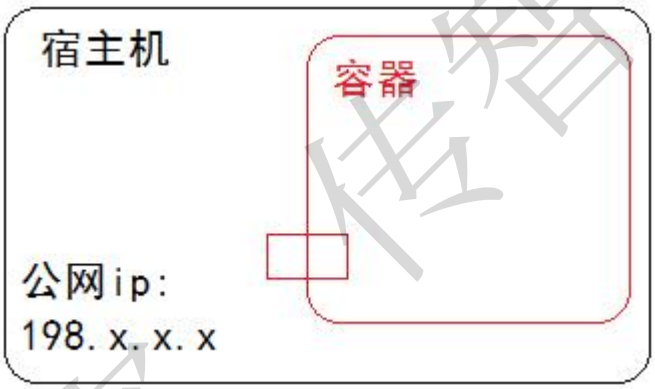
新创建的容器间使用使用已创建的容器网络，类似一个局域网。

特点：容器和容器共享网络



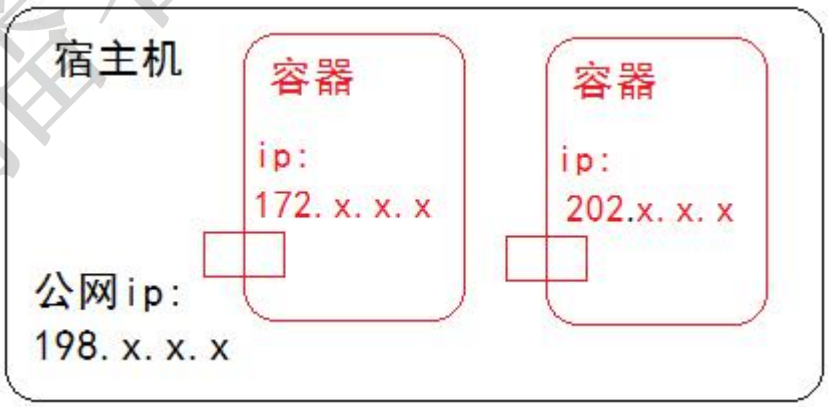
none模式：

这种模式最纯粹，不会帮你做任何网络的配置，可以最大限度的定制化。



overlay模式：

容器彼此不再同一网络，而且能互相通行。



3.6.5 bridge实践

这一节，我们从创建网络、使用网络、连接操作三个方面来学习。

其实我们在端口映射的部分就是bridge模式的简单演示了，因为他们使用的是默认bridge网络模式，现在我们来自定义桥接网络。

这一部分我们从三个方面来演示：

创建桥接网络

使用自定义网络创建容器

容器断开、连接网络

创建网络

命令格式：

```
docker network create --driver [网络类型] [网络名称]
```

命令演示：

```
docker network create --driver bridge bridge-test
```

查看主机网络类型

```
~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
8a18574f0f27        bridge              bridge              local
172ae1f1f3f5        bridge-test         bridge              local
...
```

查看新建网络的网络信息

```
~# docker network inspect bridge-test
[
    {
        "Name": "bridge-test",
        ...
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
             ...
```

宿主机又多出来一个网卡设备

```
~# ifconfig
br-172ae1f1f3f5 Link encap:Ethernet  HWaddr 02:42:18:4e:ac:92
          inet addr:172.18.0.1  Bcast:172.18.255.255  Mask:255.255.0.0
          ...
```

在自定义网络中启动容器

命令格式：

```
docker run --net=[网络名称] -itd --name=[容器名称] [镜像名称]
```

使用效果：

```
docker run --net=bridge-test -itd --name=nginx-new-bri nginx
```

查看该容器的ip信息

```
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' nginx-new-bri
```

注意部分

使用默认的桥接模型创建的容器是可以直接联网的。

使用自定义的桥接模型创建的容器不可以直接联网，但是可以通过端口映射来实现联网

容器断开网络

命令格式：

```
docker network disconnect [网络名] [容器名]
```

命令演示：

```
docker network disconnect bridge-test nginx-new-bri
```

效果展示：

```
docker network inspect bridge-test
```

容器连接网络

命令格式：

```
docker network connect [网络名] [容器名]
```

命令演示：

```
docker network connect bridge-test nginx-new-bri
```

效果展示：

```
docker network inspect bridge-test
```

3.6.6 host模型实践

这一节，我们从命令详解、host特点两个方面来学习。

命令详解

host模型我们知道，容器使用宿主机的ip地址进行对外提供服务，本身没有ip地址。

命令格式：

```
docker run --net=host -itd --name [容器名称] 镜像名称
```

命令示例：

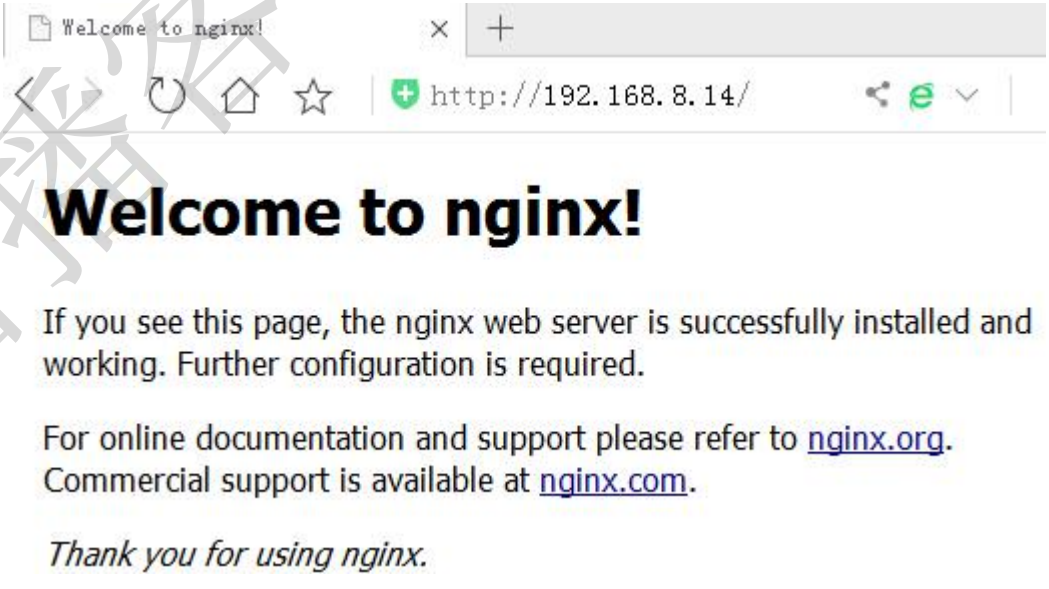
创建容器，使用host模式

```
docker run --net=host -itd --name nginx-1 nginx
```

查看宿主机端口

```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      64198/nginx: maste
```

查看网络连接

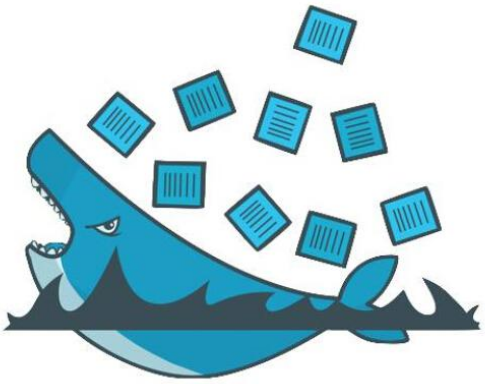


host特点：

host模型比较适合于，一台宿主机跑一个固定的容器，比较稳定，或者一个宿主机跑多个占用不同端口的应用的场景，他的网络性能是很高的。

hosts模型启动的容器不会有任何地址，他其实是使用了宿主机的所有信息

第 4 章 Docker 进阶



在这一部分我们主要来介绍一些Docker的高级内容：

Dockerfile 和 Docker compose

4.1 Dockerfile

学习目标：

了解 Dockerfile简介及特点

应用 Dockerfile使用命令

说出 Dockerfile常见指令及其特点

4.1.1 Dockerfile简介

这一节，我们从定义、作用、准则、文件内容、基础指令、使用命令这六个方面来学习。

什么是Dockerfile

Dockerfile类似于我们学习过的脚本，将我们在上面学到的docker镜像，使用自动化的方式实现出来。

Dockerfile的作用

1、找一个镜像： ubuntu

2、创建一个容器： docker run ubuntu

3、进入容器： docker exec -it 容器 命令

4、操作： 各种应用配置

....

5、构造新镜像： docker commit

Dockerfile 使用准则

1、大： 首字母必须大写D

2、空： 尽量将Dockerfile放在空目录中。

3、单： 每个容器尽量只有一个功能。

4、少： 执行的命令越少越好。

Dockerfile 基础四指令：

基础镜像信息	从哪来？
维护者信息	我是谁？
镜像操作指令	怎么干？

容器启动时执行指令　　嗨！！！

Dockerfile使用命令：

构建镜像命令格式：

```
docker build -t [镜像名]:[版本号] [Dockerfile所在目录]
```

构建样例：

```
docker build -t nginx:v0.2 /opt/dockerfile/nginx/
```

参数详解：

-t　　指定构建后的镜像信息，

/opt/dockerfile/nginx/　　则代表Dockerfile存放位置，如果是当前目录，则用 .（点）表示

4.1.2 Dockerfile快速入门

这一节，我们从环境、文件、构建、效果这四个方面来速的使用Dockerfile来创建一个定制化的镜像：ssh。

准备环境

创建Dockerfile专用目录

```
mkdir /docker/images/ssh -p
cd /docker/images/ssh
```

创建秘钥认证

```
ssh-keygen -t rsa
cat ~/.ssh/id_rsa.pub > authorized_keys
```

准备软件源

```
cp /etc/apt/sources.list ./
```

注意：

课堂上因为多方面的原因，我们不执行这一步-20181125

sources.list

定制文件

创建Dockerfile文件

```
# 构建一个基于 ubuntu 的 ssh 定制镜像
# 基础镜像
FROM ubuntu-base
# 镜像作者
MAINTAINER President.Wang 000000@qq.com

# 执行命令
# 增加软件源  -- 由于课堂网络原因，我们不执行这一步
ADD sources.list /etc/apt/sources.list
# 安装 ssh 服务
RUN apt-get update && apt-get install -y openssh-server curl vim net-tools && mkdir -p /var/run/sshd && mkdir -p /root/.ssh && sed -i "s/.*pam_loginuid.so/#&/" /etc/pam.d/sshd && apt-get autoclean && apt-get clean && apt-get autoremove
# 复制配置文件到相应位置,并赋予脚本可执行权限
ADD authorized_keys /root/.ssh/authorized_keys

# 对外端口
EXPOSE 22

# 启动 ssh
```

```
ENTRYPOINT ["/usr/sbin/sshd","-D"]
```

构建镜像

构建镜像

```
docker build -t ubuntu-ssh .
```

效果查看

使用新镜像启动一个容器，查看效果

```
docker run -d -p 10086:22 ubuntu-ssh
```

容器检查

```
docker ps
docker port c03d146b64d4
```

ssh查看效果

```
ssh 192.168.8.14 -p 10086
```

4.1.3 基础指令详解

这一节，我们来学习五个基础指令

基础指令

FROM

```
格式：
    FROM <image>
    FROM <image>:<tag>。
解释：
    FROM 是 Dockerfile 里的第一条而且只能是除了首行注释之外的第一条指令
```

MAINTAINER

```
格式：
    MAINTAINER <name>
解释：
    指定该 dockerfile 文件的维护者信息。类似我们在 docker commit 时候使用-a 参数指定的信息
```

RUN

```
格式：
    RUN <command>                                    (shell 模式)
    RUN["executable", "param1", "param2"]。          (exec 模式)
解释：
  表示当前镜像构建时候运行的命令
注释：
  shell 模式：类似于  /bin/bash -c command
              举例：  RUN echo hello
  exec  模式：类似于 RUN ["/bin/bash", "-c", "command"]
              举例：  RUN ["echo", "hello"]
```

EXPOSE

```
格式：
  EXPOSE <port> [<port>...]
解释：
   设置 Docker 容器对外暴露的端口号，Docker 为了安全，不会自动对外打开端口，如果需要外部提供访问，还需要启动容器时增加-p 或者-P 参数对容器的端口进行分配。
```

ENTRYPOINT

```
格式：
  ENTRYPOINT ["executable", "param1","param2"]          (exec 模式)
  ENTRYPOINT command param1 param2                      (shell 模式)
解释：
   每个 Dockerfile 中只能有一个 ENTRYPOINT，当指定多个时，只有最后一个起效。
```

4.1.4 文件编辑指令详解

这一节，我们从指令详解、ADD实践、COPY实践、VOLUME实践这四个方面来学习。

注意：

ADD和COPY相当于数据卷操作，VOLUME相当于数据卷容器操作

目录文件编辑指令

ADD

```
格式：
   ADD <src>... <dest>
   ADD ["<src>",... "<dest>"]
解释：
     将指定的 <src> 文件复制到容器文件系统中的 <dest>
     src 指的是宿主机，dest 指的是容器
   如果源文件是个压缩文件，则 docker 会自动帮解压到指定的容器中(无论目标是文件还是目录，都会当成目录处理)。
```

COPY

```
格式：
   COPY <src>... <dest>
   COPY ["<src>",... "<dest>"]
解释：
    单纯复制文件场景，Docker 推荐使用 COPY
```

VOLUME

```
格式：
   VOLUME ["/data"]
解释：
  VOLUME 指令可以在镜像中创建挂载点，这样只要通过该镜像创建的容器都有了挂载点
  通过 VOLUME 指令创建的挂载点，无法指定主机上对应的目录，是自动生成的。
   举例：
     VOLUME ["/var/lib/tomcat7/webapps/"]
```

ADD实践

拷贝普通文件

Dockerfile文件内容

```
...
# 执行命令
...
# 增加文件
ADD ["sources.list","/etc/apt/sources.list"]
...
```

拷贝压缩文件

Dockerfile文件内容

```
...
```

```
# 执行命令
...
# 增加文件
ADD ["linshi.tar.gz","/nihao/"]
...
```

COPY实践

修改Dockerfile文件内容：

```
...
# 执行命令
...
# 增加文件
COPY index.html /var/www/html/
...
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
```

VOLUME实践

修改Dockerfile文件内容：

```
# 在上一个 Dockerfile 文件内容基础上，在 COPY 下面增加一个 VOLUME
VOLUME ["/data/"]
...
```

4.1.5 环境指令详解

这一节，我们从指令详解、ENV实践、WORKDIR实践这三个方面来学习。

环境设置指令

ENV

```
格式：
    ENV <key> <value>
    ENV <key>=<value> ...
解释：
    设置环境变量，可以在 RUN 之前使用，然后 RUN 命令时调用，容器启动时这些环境变量都会被指定
```

WORKDIR

```
格式：
    WORKDIR /path/to/workdir                    (shell 模式)
解释：
    切换目录，为后续的 RUN、CMD、ENTRYPOINT 指令配置工作目录。 相当于 cd
    可以多次切换(相当于 cd 命令)，
    也可以使用多个 WORKDIR 指令，后续命令如果参数是相对路径，则会基于之前命令指定的路径。例如
    举例：
    WORKDIR /a
    WORKDIR b
    WORKDIR c
    RUN pwd
    则最终路径为 /a/b/c。
```

ENV实践

修改Dockerfile文件内容：

```
# 在上一个 Dockerfile 文件内容基础上，在 RUN 下面增加一个 ENV
ENV NIHAO=helloworld
```

WORKDIR实践

修改Dockerfile文件内容：

```
# 在上一个 Dockerfile 文件内容基础上，在 RUN 下面增加一个 WORKDIR
WORKDIR /nihao/itcast/
RUN ["touch","itcast.txt"]
```

4.1.6 Dockerfile构建过程

这一节，我们从构建过程、镜像介绍、构建缓存这三个方面来学习。

Dockerfile构建过程：

从基础镜像1运行一个容器A
遇到一条Dockerfile指令，都对容器A做一次修改操作
执行完毕一条命令，提交生成一个新镜像2
再基于新的镜像2运行一个容器B
遇到一条Dockerfile指令，都对容器B做一次修改操作
执行完毕一条命令，提交生成一个新镜像3
...

构建过程镜像介绍

构建过程中，创建了很多镜像，这些中间镜像，我们可以直接使用来启动容器，通过查看容器效果，从侧面能看到我们每次构建的效果。

提供了镜像调试的能力

我们可以通过docker history <镜像名> 来查看整个构建过程所产生的镜像

拓展：

执行的步骤越多越好呢？还是越少越好？

构建缓存

我们第一次构建很慢，之后的构建都会很快，因为他们用到了构建的缓存。

不适用构建缓存方法常见两种：

全部不同缓存：

```
docker build --no-cache -t [镜像名]:[镜像版本] [Dockerfile位置]
```

部分使用缓存：

```
ENV REFRESH_DATE 2018-01-12
```

只要构建的缓存时间不变，那么就用缓存，如果时间一旦改变，就不用缓存了

样例：

```
# 构建一个基于 ubuntu-base 的 docker 定制镜像
# 基础镜像
FROM ubuntu-base

# 镜像作者
MAINTAINER President.Wang 000000@qq.com

# 创建构建刷新时间
ENV REFRESH_DATE 2018-11-02
```

```
# 执行命令
...
```

构建历史：

查看构建过程查看

```
docker history
```

清理构建缓存：

```
docker system prune
docker system prune --volumes
```

4.2 Dockerfile构建django环境

学习目标：

应用 Dockerfile的生产使用流程

应用 Dockerfile定制web项目镜像

4.2.1 项目描述

django官方网站：https://code.djangoproject.com/

基于我们在shell自动化运维课程中的项目案例，我们现在来使用Dockerfile做一个Docker镜像。

4.2.2 手工部署django项目环境

这一节，我们从需求、方案分析、技术关键点、方案、方案实施这五个方面来学习。

需求：

基于docker镜像，手工部署django项目环境

方案分析：

1、docker环境部署

2、django环境部署

3、djanog项目部署

4、测试

技术关键点：

1、docker环境部署

使用docker镜像启动一个容器即可

2、django环境部署

django软件的依赖环境

django软件的基本环境配置

3、django项目部署

django框架的下载

项目文件配置

启动django项目

4、测试

宿主机测试

解决方案：

1、docker环境配置

1.1 获取docker镜像

1.2 启动docker容器

2、django环境部署
2.1 基础环境配置
2.2 django环境配置
3、django项目部署
3.1 创建django项目
3.2 创建django应用
3.3 项目启动
4、测试
4.1 宿主机测试

方案实施：

1、docker环境配置

1.1 获取docker镜像

```
ubuntu-ssh
```

1.2 启动docker容器

启动容器，容器名称叫 django

```
docker run -d -p 10086:22 --name django ubuntu-ssh
```

进入容器

```
ssh 192.168.8.14 -p 10086
```

2、django环境部署

2.1 基础环境配置

基础目录环境

```
mkdir /data/{server,softs} -p
cd /data/softs
scp root@192.168.8.14:/data/softs/Django-2.1.2.tar.gz ./
```

注意：

因为我们的docker镜像就是唯一的，所以就没有必要单独创建python虚拟环境了

2.2 django环境配置

安装基本依赖软件

```
apt-get install python3-pip -y
```

安装django软件

```
cd /data/softs
tar xf Django-2.1.2.tar.gz
cd Django-2.1.2
python3 setup.py install
```

3、django项目部署

3.1 创建django项目

创建项目

```
cd /data/server
django-admin startproject itcast
```

3.2 创建django应用

创建应用

```
cd /data/server/itcast
python3 manage.py startapp test1
```

注册应用

```
# vim itcast/settings.py

INSTALL_APP = [
    。。。
```

```
    'test1',
]
```

配置应用

```
admin-1@ubuntu:/data/soft# cat /data/server/itcast/test1/views.py
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.

def hello(resquest):
   return HttpResponse("itcast V1.0")
```

url文件配置

```
admin-1@ubuntu:/data/soft# cat /data/server/itcast/itcast/urls.py
...
from test1.views import *

urlpatterns = [
     path('admin/', admin.site.urls),
     path('hello/', hello),
]
```

3.3 项目启动

设置访问主机

```
# vim itcast/settings.py
...
ALLOWED_HOSTS = ['*']
```

启动项目

```
python3 manage.py runserver 172.17.0.2:8000
```

注意：

要提前用ifconfig来检查一下当前容器的IP地址，以方面接下来的测试

4、测试

4.1宿主机测试

查看容器的ip地址

```
docker inspect django
```

浏览器、或curl查看效果：

```
172.17.0.2:8000/hello/
```

4.2.3 Dockerfile案例实践

这一节，我们从环境分析、关键点分析、定制方案、Dockerfile实践这三个方面来学习

环境分析：

1、软件源文件，使用国外源，速度太慢，所以我们可以自己使用国内的软件源。

因为我们在手工部署的时候，使用的是官方(国外)的源，所以为了部署快一点呢，我使用国内的源。

2、软件安装，涉及到了各种软件

3、软件运行涉及到了软件的运行目录

4、项目访问，涉及到端口

关键点分析：

1、增加文件，使用 ADD 或者 COPY 指令
2、安装软件，使用 RUN 指令
3、命令运行，使用 WORKDIR 指令
4、项目端口，使用 EXPOSE 指令
5、容器运行，使用 ENTRYPOINT

定制方案：

1、基于ubuntu-ssh基础镜像进行操作
2、安装环境基本软件
3、定制命令工作目录，并增加文件
4、开放端口
5、执行项目

Dockerfile定制

进入标准目录

```
mkdir /docker/images/django -p
cd /docker/images/django
```

第一版dockerfile内容

```
# 构建一个基于 ubuntu 的 docker 定制镜像
# 基础镜像
FROM ubuntu-ssh
# 镜像作者
MAINTAINER President.Wang 000000@qq.com
# 执行命令
RUN apt-get install python3-pip -y
# 增加文件
ADD Django-2.1.2.tar.gz /data/softs/
WORKDIR /data/softs/Django-2.1.2
RUN python3 setup.py install

# 创建项目
WORKDIR /data/server
RUN django-admin startproject itcast
# 创建应用
WORKDIR /data/server/itcast
RUN python3 manage.py startapp test1
RUN sed -i "/staticfiles/a\    'test1'," itcast/settings.py
# 配置应用
COPY views.py /data/server/itcast/test1/
RUN sed -i '/t p/a\from test1.views import *' itcast/urls.py
RUN sed -i "/\]/i\    path('hello/', hello)," itcast/urls.py
# 启动项目
RUN sed -i "s#S = \[\]#S = \['*'\]#" itcast/settings.py
# 对外端口
EXPOSE 8000
# 运行项目
ENTRYPOINT ["python3","manage.py","runserver","0.0.0.0:8000"]
```

把Django-2.1.2.tar.gz和views.py文件放到这个目录中

效果查看

构建镜像

```
docker build -t ubuntu-django .
```

运行镜像

```
docker run -p 8000:8000 -d ubuntu-django
```

访问镜像，查看效果

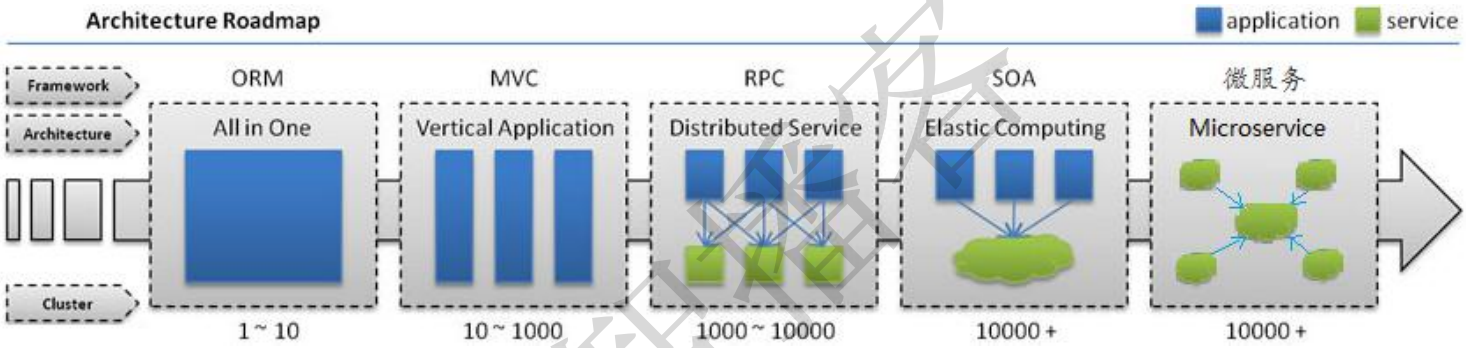
第 5 章 部署串讲

5.1 部署项目

从这一节开始，我们就来说一下和项目部署相关的知识体系。

5.1.1 框架演变

项目开发框架演变



5.1.2 架构演变

这一节的目的就是让大家来了解一下，互联网的软件项目架构一般的发展过程。

项目的架构

一般来说，任何一个项目至少有三层内容来组成：

web访问层、数据库层、存储层

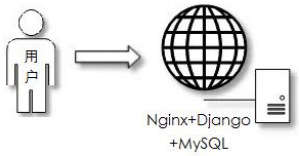
初级阶段

单体阶段

常见场景：项目初期

部署特点：所有应用服务都在一台主机

应用特点：开发简单

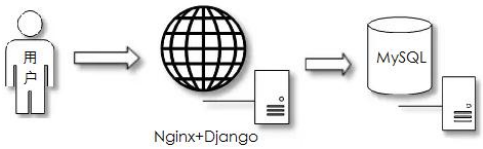


应用/数据分离阶段

常见场景：项目初期，用户访问数据库有压力

部署特点：应用和数据库单独部署

应用特点：开发简单

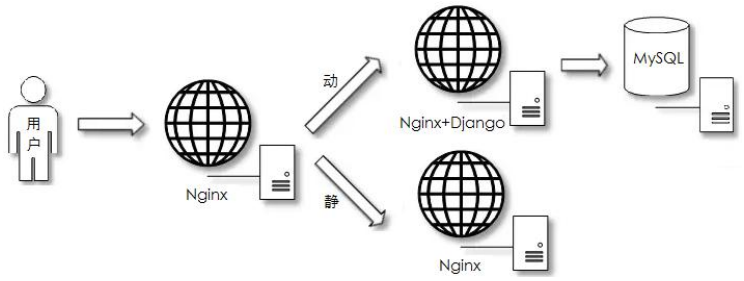


页面动静分离阶段

常见场景：项目初期，用户访问页面有压力

部署特点：剥离用户读请求和写请求操作

应用特点：开发简单

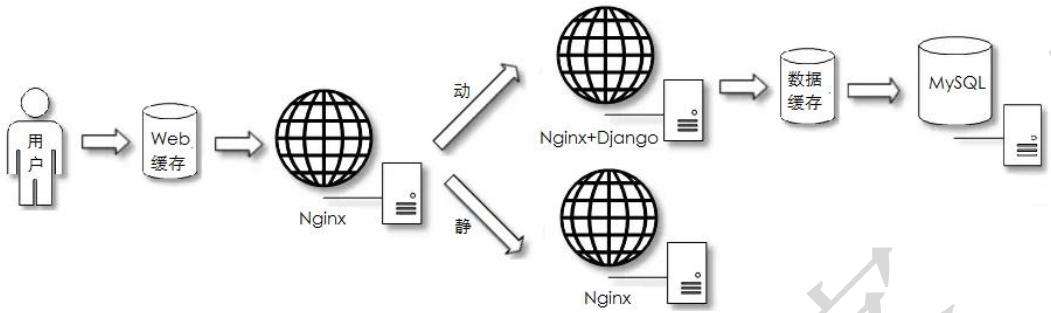


页面/数据缓存阶段

常见场景：项目初期，用户访问有压力

部署特点：代理和数据库前面增加缓存组件

应用特点：开发简单



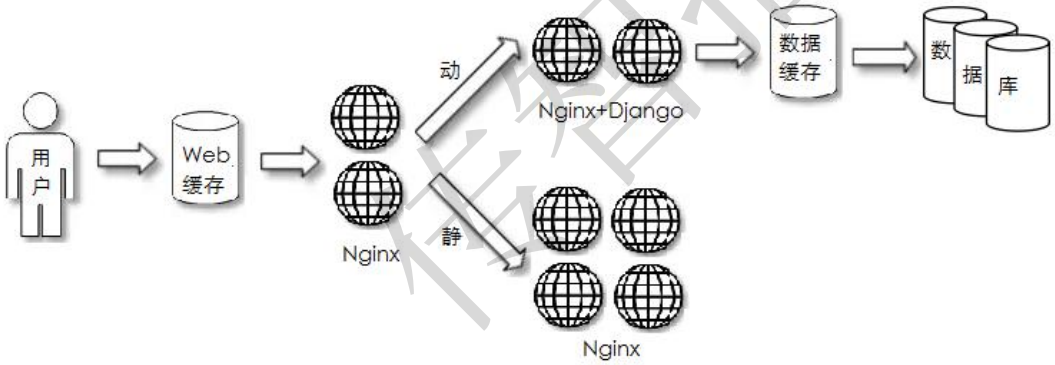
中期阶段

应用服务集群阶段

常见场景：项目初期，用户访问有压力

部署特点：应用服务所在主机做集群负载均衡

应用特点：业务中等

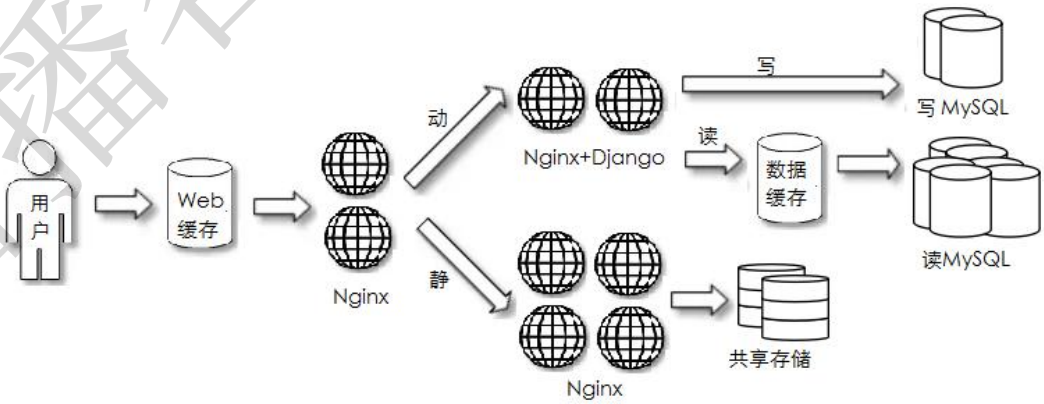


数据库读写分离化

常见场景：项目初期，用户访问数据有压力

部署特点：对数据库集群做读写分离，静态文件做共享存储

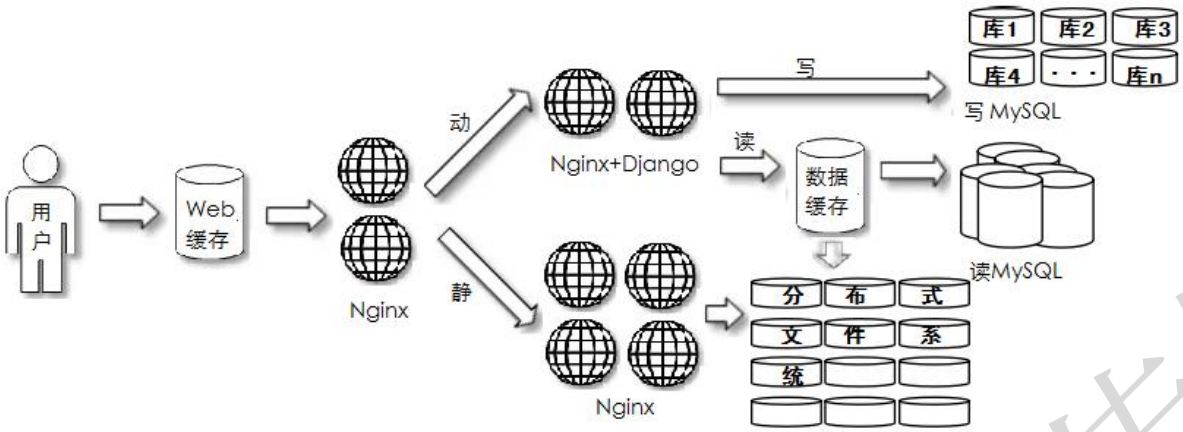
应用特点：业务中等



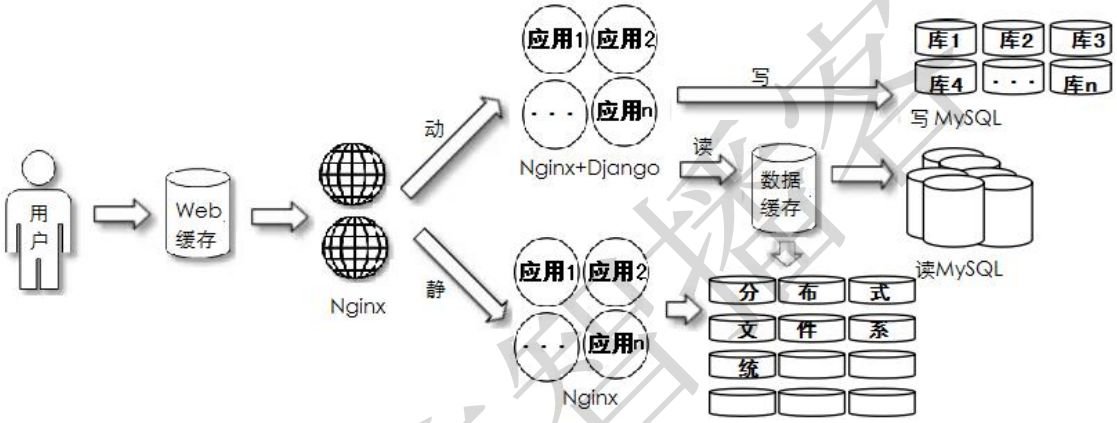
存储分布式

常见场景：项目中期，数据存储有压力

部署特点：对数据库分库/分表扩展，数据文件使用分布式存储
应用特点：业务中等

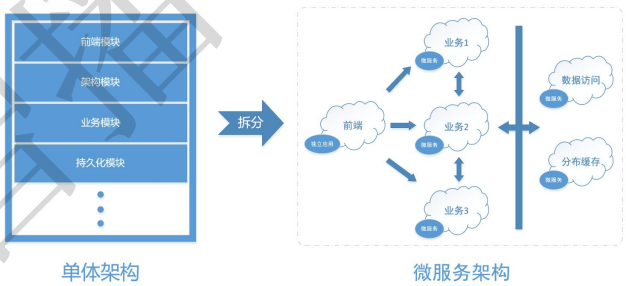


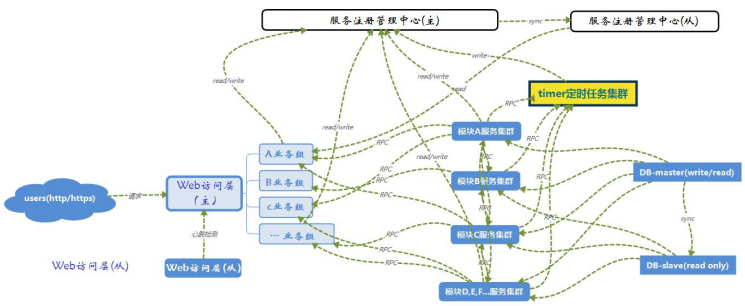
业务应用拆分
常见场景：项目中期，业务访问/团队管理有压力
部署特点：项目应用进行拆分
应用特点：业务复杂



中后期阶段
业务拆分
常见场景：项目中后期，业务处理有压力
部署特点：所有功能以服务形式单独部署，引入配置管理管理中心、消息中间件，搜索引擎等功能
应用特点：业务复杂

微服务阶段
常见场景：项目后期，精益求精
部署特点：所有服务都可以自由部署
应用特点：业务复杂





5.1.2 架构部署

通用架构

一级定位：核心组成部分

web 、数据库、存储层

二级定位：功能增强部分

web缓存、代理、数据库缓存

部署项目

部署项目的时候，要遵循主次原则：

对于架构层中的一级角色，我们的部署原则是：

站在用户访问资源角度，从后向前依次部署。

对于架构层中的二级角色，我们的部署原则是：

站在用户访问资源压力角度，需要部署哪里，就部署哪里，注意前后的信息交流。

5.2 项目运营

在项目正常运营过程中，为了验证我们项目的效果和更好的对项目的运行维护，我们必须知道一些网站分析和运营维护方面的基础知识

5.2.1 网站分析

常见术语

我们在日常生活中经常会听说，XX网站日PV多少，日UV多少，每日访问峰值是多少之类的话，而这些名词都是项目正常运营的过程中，为了更好的对项目的运行维护，我们对网站进行分析必须知道一些常用的网站分析术语。

IP：独立IP数

指一天内使用不同IP地址的用户访问网站的数量。

特点：

同一个IP无论访问多少网页，独立IP数均为1

PV：Page view页面浏览量

指一天内网站的浏览次数，它是衡量网站用户访问页面的数量。

特点：

用户每打开一个页面就记录一次，就算访问同一页面多次，就记录多次

UV：Unique Visitor 访问网站的用户

指一天内访问某站点的人数，以cookie/客户端为依据

特点：

一天内，同一访问用户的多次访问只记录1次

VV：Visit View 用户访问网站次数

指一天内某个用户访问了多少次网站

特点：

打开网页A，浏览完毕后关闭该页面，表示一次访问

BR：Bounce Rate 跳出率

指一天内访问用户中，用户打开网站后没有做任何事情，一会就离开了的比例

特点：

如果跳出率很高，说明我们的网页没有什么吸引力，网页设计效果不怎么好

CR：Conversion Rate 转化率

指一天内访问用户中，打开网站后，继续浏览该网站其他页面的比例

特点：

转化率一般体现在项目的关键流程的部分，而它对网站的某些关键流程优化是一个很重要的直播

网站分析术语多如牛毛，每个公司的业务方向不一样，评判标准也不一样，上面只是举例了几种常见的术语，这些术语的常用场景如下：

口语描述：

术语：IP、PV、UV

人员：所有岗位

网站质量：

术语：跳出率和转化率

人员：产品、开发

常见分析工具

服务器服务日志、公司内部监控平台、等

互联网网站分析工具：

站长工具、百度统计、云平台监控等

5.2.2 项目运营

项目在后期运营的过程中主要做两件事情：项目正常运行、项目功能迭代

项目正常运行

关于项目正常运行，就是网站运行过程中，不论遇到什么问题，我们都能应对下去。一般来说就是用户访问量变化的时候我们做的优化等工作。那么我们就站在开发的角度，从网站的分层上面来看一下常见问题及其解决方案。

缓存层方面

问题描述：

怎么在现有的主机资源情况下，花最小的代价抗住大量的用户访问量？

解决思路：

很常见的就是自建Web缓存，或者购买CDN，将用户经常访问的、更新频率低的资源存放起来，这样用户再次请求相同资源的时候，就不会对后端的服务造成影响。当然防止互联网上的恶意访问/爬虫，我们应该做好相应的安全措施。

缓存之类的措施一定要适合公司的当前业务，如果是项目的静态资源很多，只要我们购买的CDN够好，那么用户访问量随便。

代理层方面

问题描述：

如何提高用户高质量的请求分发。

解决思路：

以Nginx代理为例，提高用户高质量的请求分发，最好的方法就是基于请求的关键字进行合理的分流。

基于请求的IP地址信息，封闭恶意IP访问，提高正常IP用户访问效率

基于请求的浏览器信息，分发到相应的后端应用，

基于请求的协议方法，做好读写分离业务的精确分流，

基于请求的路径信息，做好指定业务的精确分流，

问题描述：

对于前端使用nginx进行代理的项目，会随着功能的层层迭代逐渐增加数十个upstream，location规则的数量有可能达到数百个，这个时候偶尔有些 URL 会出现 404 的问题，对于这种情况怎么办？

解决思路：

1 按照功能描述，将upstream拆分到不同的功能目录中

2 对location的匹配规则尽量描述清楚，特别是匹配的location_match，最好使用^/$来锚定首尾字母

项目后端web访问

问题描述：

关于动态web请求过多，压力有些大，常见的解决方法有哪些？

解决思路：

可以根据架构演变的思路，我们合理的调整页面访问的关键流程，在技术方面我们可以这么做：

1 分析动态的web请求主要的瓶颈点在哪里，是请求量大，还是数据访问大

请求量大：

Web缓存/CDN，或者动态web集群可以考虑一下

数据库操作多：

分析请求内容是否频繁/集中，是，页面静态化考虑一下；否，参看数据库的演变思路

问题描述：

如何提高静态web资源的访问质量？

解决思路：

结合前段缓存的功能，在代码或者代理部分设置合理的资源缓存过期时间，定时/实时推送相关信息到前段的缓存层。

数据层方面

问题描述：

用户访问数据有压力

解决思路：

对于热点数据读取频繁的话，可以考虑前端数据缓存、分部署数据缓存、优化查询搜索等方法

对于数据频繁写入压力的话，可以考虑数据库集群、读写分离、分库分表、增加数据管理层等方法

开发角度：

关注数据库表的设计，表的索引合理、查询的时候，尽量使用条件查询

存储层方面：

问题描述：

如何保证我们数据存储的质量

解决思路：

存储设备的购买质量、分布式存储、备份策略

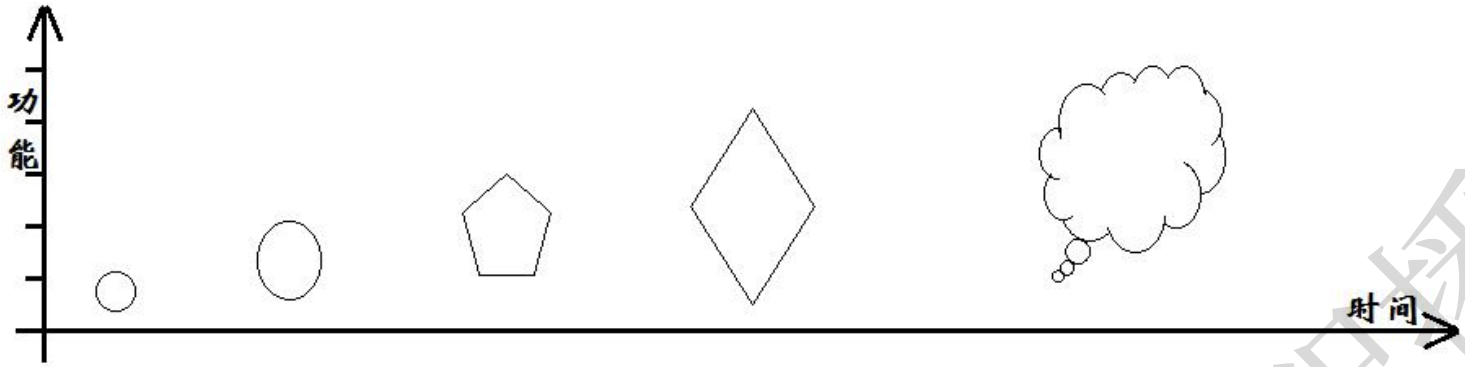
5.3 功能迭代

5.3.1 边做边改模型

边做边改模型 (Build-and-Fix Model)

边做边改模型很常见，很多公司由于项目、资金、人员、投资等方面的原因，无限制的压缩产品的交付周期，根据制定的临时需求计划[几乎没有规划]就进行项目开发，通过快速的创建调试后，交付给用户产品，如果运行起来后，不满足用户需求，那么就在当前产品的基础上修改，直到达到指定用户的满意为止。

简单来说就是：走一步，看一步，再走一步，...



优势：

这种作坊式的软件生产方法，好处就是快速的产出效益，企业回笼资金快。

劣势：

1 野草式发展：无计划的发展，随着软件结构的无规律修改，产品会越来越糟，最终导致无法继续修改。

2 质量无保证：开发过程无考虑测试和可维护性，也没有任何文档，软件的维护十分困难。

场景：

项目初期使用最频繁

逻辑不需要太严谨的小项目

5.3.2 瀑布模式

瀑布模式(Waterfall Model)

瀑布模型提供了软件开发的基本框架。它在传统的软件生命周期的基础上，以文档计划来推动项目生产的一种软件开发模式。其过程是：每个阶段(B)只接受上一阶段(A)的工作成果(A-Pro)，同时评审该内容是否适合当前阶段(B)工作的基本要求。

若不符合，则返回前一阶段(A)，甚至更前面的阶段进行确认；

若符合，则继续完成当前阶段(B)的工作内容,并将生成工作成果(B-Pro)。然后继续刚才"接受-验证-继续"的循环，直到产品完成交给用户。

优势：

因为瀑布模型的每个阶段的边界划分很清楚，而且以线性的顺序依次完成。如果我们将阶段间的联系划分的更细一些，意味着我们在阶段性交接工作成果的时候没有更多的验证约束。即使交接的工作成果有问题，我们只需要纠正一下即可。

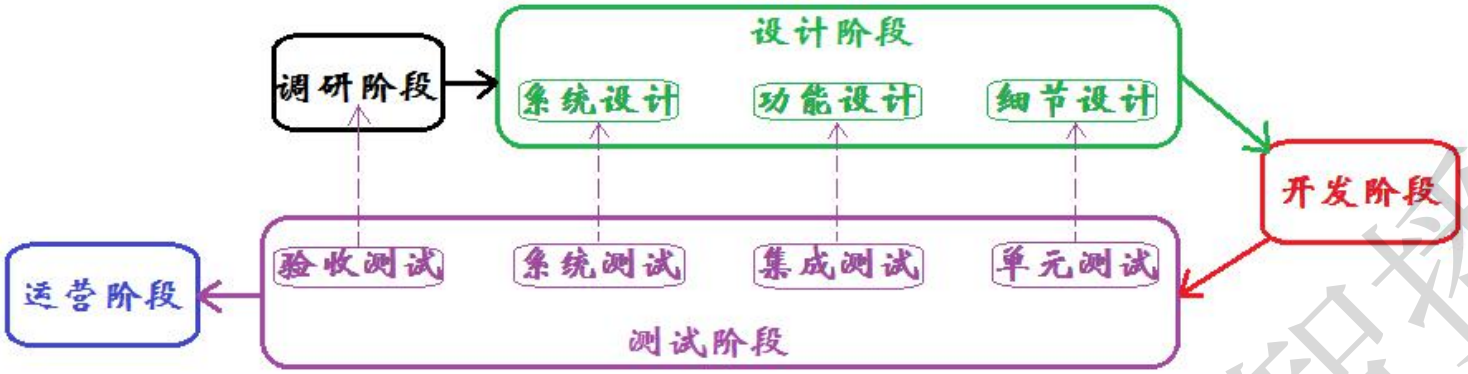
瀑布模型有明确的需求分析，层次性的开发思路，我们只需要按部就班的完成各阶段的工作内容就可以，它是一种理想化的开发模型，非常适合过程管理。

劣势：直接影响正常的产品交付日期

瀑布模型的优势也是劣势。按计划完成任务很好，但是真正的工作中有非常多的不确定性。

1 需求不确定：因为我们的分析阶段是自己调研的，而用户在真正接触到产品前，需求会发生变化，而且开发过程中会有很多的不确定因素，导致我们不能按照理想化的路走下去。

2 反馈太慢长：我们按照计划开发产品，A阶段完成后，只有到B阶段的时候才会检查A阶段工作结果的效果，效果反馈慢。而且有些错误是不可避免的，因为需求分析阶段是最容易出现错误的阶段，也就是说，我们会在错误的道路上走到最后(部署出来)才发现错了(需求错误)，这就意味着，当我们意识到错误的时候，已经迟了，我们需要花费很大的成本来修复错误。



3 缺乏并行性：每个阶段只接受上一阶段的成果后才工作，没有办法多阶段并行工作。

4 效率低下：A阶段在进行过程中，剩余的阶段都在等待，大量的资源(人力、物力)可能处于闲置状态，而且工作的人技术不一定很好。

场景：

大部分的项目初期阶段

某些核心、大型、稳定性要求高的项目

5.3.3 迭代模型

迭代模型 (stagewise model)

瀑布模型出现的一段时间内，很多政府部门、企业都采用这种方式来开发自己的产品，由于产品规模越来越大，产品交付时间越来越短等因素，导致瀑布模型的劣势越来越突出，远远超出其创造的价值。这些劣势主要集中在不可控需求风险及产品交付周期上。

迭代模型是为了实现快速的将产品交付给用户，在设计产品的时候，不像瀑布模型那样设计的非常大/完美，而是一个阶段一个阶段的实现部分功能，最终交付给用户一个完善的产品。其中每一个阶段的功能都使用瀑布模型开发，并且有一个可交付的产品成果。

优势：

1 反馈周期短：每个阶段的工作成果可以快速的交付给用户，用户接到产品后，使用的效果，都可以快速的反馈给产品人员。

2 降低产品风险：开发工作按照既定的计划进行推进，而且推进的过程中，可以结合上一个阶段的用户反馈来细化需求或者合理的变动部分功能/业务逻辑,并开始新一轮的迭代.

3 提高效率：阶段性的功能拆分及快速质量反馈，是的开发人员清楚产品的功能定位和问题焦点，工作效率会提高，加快整个项目工作的进度。

劣势：

团队水平：项目研发过程中，功能需求变动频繁导致风险增多，这对领导/组织者水平要求要高一些，软件研发团队的综合应变水平也有一定的要求。

场景：

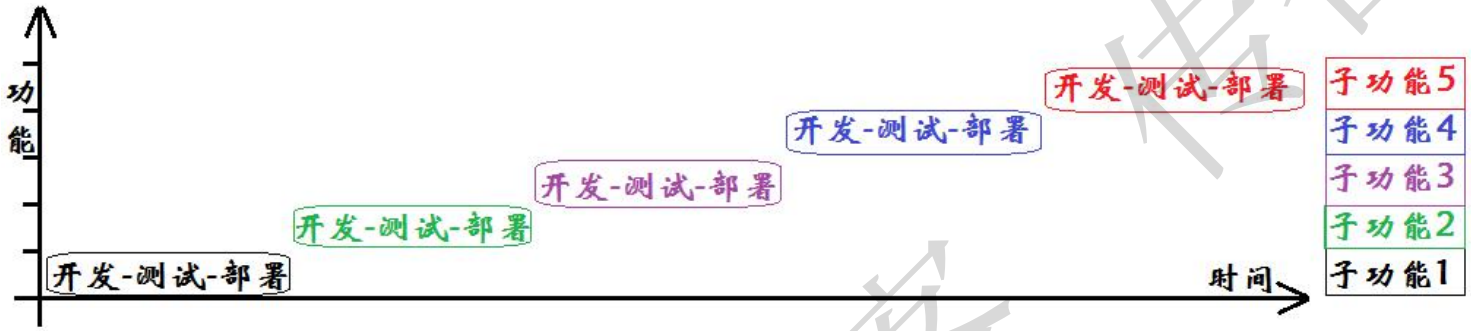
大部分项目的中后期的通用做法

高风险项目

5.3.4 增量模型

增量模型 (Incremental Model)

增量模型是一种分步开发的模型。它集成了瀑布模型的顺序特征和迭代模型的迭代特性。一般情况下，先针对一个大型的产品进行精细化设计，将复杂项目进行合理的阶段性功能拆分，然后每一个阶段的功能产品都使用瀑布模型开发，并且交付的子功能产品成果。每个阶段(B)都在前一个阶段(A)实现的功能基础上进行迭代开发，多个功能阶段迭代完毕后，就可以将最终完善的产品交付给用户了。



优势：

在保证项目目标的方向上，产品交付时间比瀑布模型短

在保证交付时间的标准上，产品功能目标比迭代模型好

劣势：

1 精细设计程度：在产品功能设计的时候，要把控好阶段性子功能的边界，对需求经常大变动的项目不太适合。

2 阶段性依赖：当前(B)阶段是前一个(A)阶段功能产品的基础上进行的，而且当前(B)阶段功能开发的过程中，不能破坏前一个(A)阶段的功能

3 团队水平：项目研发过程中，功能需求变动频繁导致风险增多，这对领导/组织者水平要求要高一些，软件研发团队的综合应变水平也有一定的要求。

场景：

大部分项目早期使用增量模型，可以规避技术风险。

交付时间紧张、人员不足的项目场景都可以。

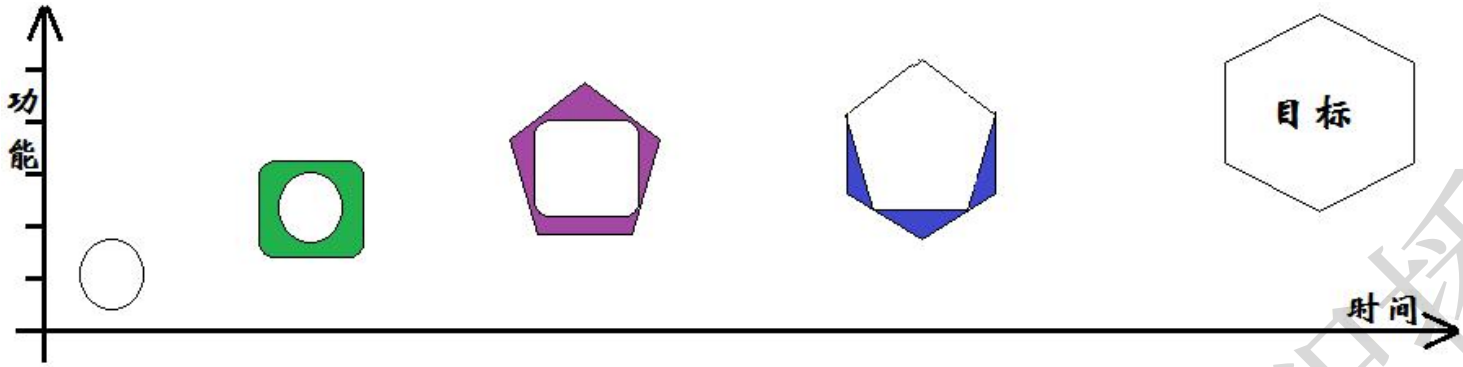
产品需求功能变动大的场景不太适合。

5.3.5 快速原型模型

快速原型模型 (Rapid Prototype Model)

快速原型这种模型一般使用于需求复杂/动态变化的软件系统。允许在调研阶段只对部分核心功能进行分析，用最少的时间做一个基本功能完备(可以运行)的产品原型，然后快速推上线，结合用户使用过程中具体的反馈信息，以丰富细化产品功能需求，在当前的产品原型基础上进行功能迭代更新,最终开发出客户满意的软件产品。

快速原型模型，又称原型模型，有点像“边做边改”与“迭代模型”的一个特殊的增量模型。以进化的开发方式为中心，迭代合理的新功能。



优势：

快速原型的目的在需求复杂不明的时候，通过快速的软件原型，理解和澄清问题，以便研发团队与用户达成共识，减少由于软件需求不明确带来的开发风险。

迭代新功能的时候会结合用户的反馈需求进行丰富细化，开发人员能确定客户的真正需求，所以该模型生产的产品一般比较能满足用户的需求。

劣势：

用户是善变的，开发会受到需求变更的影响，所以需要开发人员有快速重构项目的能力，应对各种未知风险的能力。

场景：

特殊的、需求不明的、复杂的项目，或者项目中的某些紧急bug。

5.3.6 敏捷开发模型

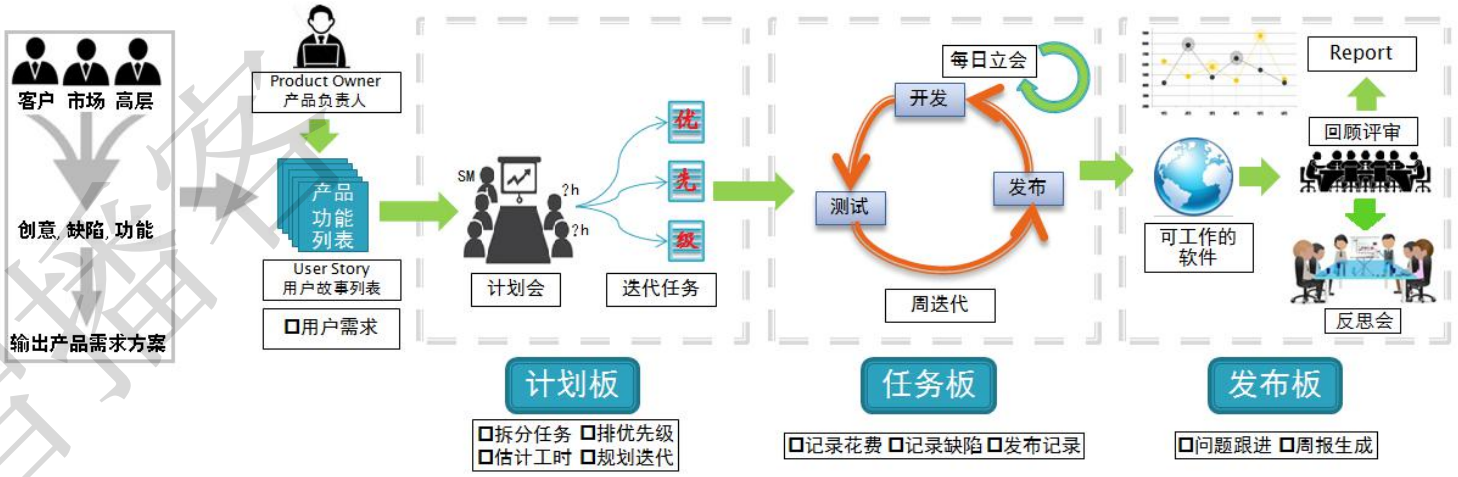
敏捷开发 (Agile Development)

敏捷开发，是一种应对快速变化的用户需求的一种开发软件的管理新模式，其实它是XP、Scrum等数十种软件开发项目管理方法的集合，主要特点是：响应变更快、关注产品价值、注重个人的能力。我们从项目开发、功能迭代、团队沟通三个方面来学习。

项目开发：在敏捷开发中，最大的特点就是软件架构的解耦。也就是说：软件项目在初期被切分成多个相互联系，但也可独立运行的小项目，并分别完成，在整个软件开发过程中产品一直处于可使用状态。

功能迭代：强调较短的开发周期提交软件产品，相较于迭代模型更短(2-4周)。

核心特征：相较于个人(单团队)完成项目的传统软件开发模式(以文档方式推动项目前进)，敏捷开发更强调团队之间的紧密协作、团队小而精干，基于面对面的沟通，制定定迭代功能的优先级，能够很好地适应需求变化。



优势：

产品团队、研发团队、测试团队之间更注重紧密协作，

团队小而精干，面对面(口头、源代码)交流，来深入理解产品的结构和功能。

开发内嵌测试，质量前置，等等

劣势：

团队的组建较难，人员不多，但综合技术能力要求较高。

场景：

项目复杂、交付周期短，功能迭代快的项目